A Concurrent-safe Centralized Pointer Managing Facility

Changkun Ou*

June 10, 2021

Abstract

In the Go 1.17 release, we contributed a new cgo facility runtime/cgo.Handle in order to help future cgo applications better and easier to build concurrent-safe applications while passing pointers between Go and C. This article will guide us through the feature by asking what the feature offers to us, why we need such a facility, and how exactly we contributed to the implementation eventually.

Starting from Cgo and X Window Clipboard

Cgo[6] is the de facto approach to interact with the C facility in Go. Nevertheless, how often do we need to interact with C in Go? The answer to the question depends on how much we work on the system level or how often do we have to utilize a legacy C library, such as for image processing. Whenever a Go application needs to use a legacy from C, it needs to import a sort of C dedicated package as follows, then on the Go side, one can simply call the `myprint` function through the imported `C` symbol:

```
/*
#include <stdio.h>

void myprint() {
    printf("Hello %s", "World");
}
*/
import "C"

func main() {
    C.myprint()
    // Output:
    // Hello World
}
```

*Email: research@changkun.de

A few months ago, while we were working on building a new package golang.design/x/clipboard[7], a package that offers cross-platform clipboard access. We found out, there is a lacking of facility in Go, despite the variety of approaches in the wild, still suffering from soundness and performance issues.

In the golang.design/x/clipboard package, we had to cooperate with cgo to access system level APIs (technically, it is an API from a legacy and widely used C system), but lacking the facility of knowing the execution progress on the C side. For instance, on the Go side, we have to call the C code in a goroutine, then do something else in parallel:

```
go func() {
    C.doWork() // Cgo: call a C function, and do stuff on C side
}()

// .. do stuff on Go side ..
```

However, under certain circumstances, we need a sort of mechanism to understand the execution progress from the C side, which brings the need of communication and synchronization between the Go and C. For instance, if we need our Go code to wait until the C side code finishes some initialization work until some execution point to proceed, we will need this type of communication precisely to understand the progress of a C function.

A real example that we encountered was the need to interact with the clipboard facility. In Linux's X window environment, clipboards are decentralized and can only be owned by each application. The ones who need access to clipboard information are required to create their clipboard instance. Say an application `A` wants to paste something into the clipboard, it has to request to the X window server, then become a clipboard owner to send the information back to other applications whenever they send a copy request.

This design was considered natural and often required applications to cooperate: If another application `B` tries to make a request, to become the next owner of the clipboard, then `A` will lose its ownership. Afterwards, the copy requests from the application `C`, `D`, and so on, will be forwarded to the application `B` instead of `A`. Similar to a shared region of memory being overwritten by somebody else and the original owner lost its access.

With the above context information, one can understand that before an application starts to "paste" (serve) the clipboard information, it first obtains the clipboard ownership. Until we get the ownership, the clipboard information will not be available for access purposes. In other words, if a clipboard API is designed in the following way:

```
clipboard.Write("some information")
```

We have to guarantee from its inside that when the function returns, the information should be available to be accessed by other applications.

Back then, our first idea to deal with the problem was to pass a channel from Go to C, then send a value through the channel from C to Go. After a quick research, we realized that it is impossible because channels cannot be passed as a value between C and Go due to the rules of passing pointers in Cgo (see a previous proposal document [4] [5]). Even there is a way to pass the entire channel value to the C, there will be no facility to send values through that channel on the C side because C does not have the language support of the `<-` operator.

The next idea was to pass a function callback, then get it called on the C side. The function's execution will use the desired channel to send a notification back to the waiting goroutine.

After a few attempt, we found that the only possible way is to attach a global function pointer and gets it called through a function wrapper:

```
/*
int myfunc(void* go_value);
*/
import "C"

// This funcCallback tries to avoid a runtime panic error when
// directly pass it to Cgo because it violates the pointer passing
// rules:
//
//   panic: runtime error: cgo argument has Go pointer to Go pointer
var (
    funcCallback   func()
    funcCallbackMu sync.Mutex
)

type gocallback struct{ f func() }

func main() {
    go func() {
        ret := C.myfunc(unsafe.Pointer(&gocallback{func() {
            funcCallbackMu.Lock()
            f := funcCallback // must use a global function variable.
            funcCallbackMu.Unlock()
            f()
        }}))
        // ... do work ...
    }()
    // ... do work ...
}
```

In above, the `gocallback` pointer on the Go side is passed through the C function `myfunc`. On the C side, there will be a call using `go_func_callback` that

being called on the C, via passing the struct `gocallback` as a parameter:

```
// myfunc will trigger a callback, c_func, whenever it is needed
// and pass the gocallback data though the void* parameter.
void c_func(void *data) {
    void *gocallback = userData;
    // the gocallback is received as a pointer, we pass it as
    // an argument to the go_func_callback
    go_func_callback(gocallback);
}
```

The `go_func_callback` knows its parameter is typed as `gocallback`. Thus a type casting is safe to do the call:

```
//go:export go_func_callback
func go_func_callback(c unsafe.Pointer) {
    (*gocallback)(c).call()
}

func (c *gocallback) call() { c.f() }
```

The function `f` in the `gocallback` is exactly what we would like to call:

```
func() {
    funcCallbackMu.Lock()
    f := funcCallback // must use a global function variable.
    funcCallbackMu.Unlock()
    f()               // get called
}
```

Note that the `funcCallback` must be a global function variable. Otherwise, it is a violation of the cgo pointer passing rules as mentioned before.

Furthermore, an immediate reaction to the readability of the above code is: too complicated. The demonstrated approach can only assign one function at a time, which is also a violation of the concurrent nature. Any per-goroutine dedicated application will not benefit from this approach because they need a per-goroutine function callback instead of a single global callback. By then, we wonder if there is a better and elegant approach to deal with it.

Through our research, we found that the need occurs quite often in the community, and is also being proposed in golang/go#37033[1]. Luckily, such a facility is now ready in Go 1.17 :)

What is `runtime/cgo.Handle`?

The new runtime/cgo.Handle provides a way to pass values that contain Go pointers (pointers to memory allocated by Go) between Go and C without breaking the cgo pointer passing rules. A `Handle` is an integer value that can represent any Go value. A `Handle` can be passed through C and back to Go, and the Go

code can use the Handle to retrieve the original Go value. The final API design is proposed as following:

```
package cgo

type Handle uintptr

// NewHandle returns a handle for a given value.
//
// The handle is valid until the program calls Delete on it.
// The handle uses resources, and this package assumes that C
// code may hold on to the handle, so a program must explicitly
// call Delete when the handle is no longer needed.
//
// The intended use is to pass the returned handle to C code,
// which passes it back to Go, which calls Value.
func NewHandle(v interface{}) Handle

// Value returns the associated Go value for a valid handle.
//
// The method panics if the handle is invalid.
func (h Handle) Value() interface{}

// Delete invalidates a handle. This method should only be
// called once the program no longer needs to pass the handle
// to C and the C code no longer has a copy of the handle value.
//
// The method panics if the handle is invalid.
func (h Handle) Delete()
```

As we can observe: cgo.NewHandle returns a handle for any given value; the method cgo.(Handle).Value returns the corresponding Go value of the handle; whenever we need to delete the value, one can call cgo.(Handle).Delete.

The most straightforward example is to pass a string between Go and C using Handle. On the Go side:

```
package main
/*
#include <stdint.h> // for uintptr_t
extern void MyGoPrint(uintptr_t handle);
void myprint(uintptr_t handle);
*/
import "C"
import "runtime/cgo"

func main() {
    s := "Hello The golang.design Initiative"
```

```
    C.myprint(C.uintptr_t(cgo.NewHandle(s)))
    // Output:
    // Hello The golang.design Initiative
}
```

The string `s` is passed through a created handle to the C function `myprint`, and on the C side:

```
#include <stdint.h> // for uintptr_t

// A Go function
extern void MyGoPrint(uintptr_t handle);
// A C function
void myprint(uintptr_t handle) {
    MyGoPrint(handle);
}
```

The `myprint` passes the handle back to a Go function `MyGoPrint`:

```
//go:export MyGoPrint
func MyGoPrint(handle C.uintptr_t) {
    h := cgo.Handle(handle)
    s := h.Value().(string)
    println(s)
    h.Delete()
}
```

The `MyGoPrint` queries the value using `cgo.(Handle).Value()` and prints it out. Then deletes the value using `cgo.(Handle).Delete()`.

With this new facility, we can simplify the previously mentioned function callback pattern much better:

```
/*
#include <stdint.h>

int myfunc(void* go_value);
*/
import "C"

func main() {

    ch := make(chan struct{})
    handle := cgo.NewHandle(ch)
    go func() {
        // myfunc will call goCallback when needed.
        C.myfunc(C.uintptr_t(handle))
        ...
    }()
```

```
    <-ch // we got notified from the myfunc.
    handle.Delete() // no need thus delete the handle.
    ...
}

//go:export goCallback
func goCallback(h C.uintptr_t) {
    v := cgo.Handle(h).Value().(chan struct{})
    v <- struct{}
}
```

More importantly, the `cgo.Handle` is a concurrent-safe mechanism, which means that once we have the handle number, we can fetch the value (if still available) anywhere without suffering from data race.

Next question: How to implement `cgo.Handle`?

First Attempt

The first attempt[2] was a lot complicated. Since we need a centralized way to manage all pointers in a concurrent-safe way, the quickest idea that comes to our mind was the `sync.Map` that maps a unique number to the desired value. Hence, we can easily use a global `sync.Map`:

```
package cgo

var m = &sync.Map{}
```

However, we have to think about the core challenge: How to allocate a runtime-level unique ID? Passing an integer between Go and C is relatively easy, what could be a unique representation for a given value?

The first idea is the memory address. Because every pointer or value is stored somewhere in memory, if we can have the information, it would be very easy to use as the ID of the value because each value has exactly one unique memory address.

To complete this idea, we need to be a little bit cautious: Will the memory address of a living value is changed at some point? The question leads to two more questions:

1. What if a value is on the goroutine stack? If so, the value will be released when the goroutine is dead.
2. Go is a garbage-collected language. What if the garbage collector moves and compacts the value to a different place? Then the memory address of the value will be changed, too.

Based on our years of experience and understanding of the runtime, we learned that the Go's garbage collector before 1.17 is always not moving and the mech-

anism is also very unlikely to change. That means, if a value is living on the heap, it will not be moved to other places. With this fact, we are good with the second question.

It is a little bit tricky for the first question: a value on the stack may move as the stack grows. The more intractable part is that compiler optimization may move values between stacks, and runtime may move the stack when the stack ran out of its size.

Naturally, we might ask: is it possible to make sure a value always be allocated on the heap instead of the stack? The answer is: Yes! If we turn it into an `interface{}`. Until 1.17, the Go compiler's escape analysis always marks the value that should escape to the heap if it is converted as an `interface{}`.

With all the analysis above, we can write the following part of the implementation that utilizes the memory address of an escaped value:

```
// wrap wraps a Go value.
type wrap struct{ v interface{} }

func NewHandle(v interface{}) Handle {
    var k uintptr

    rv := reflect.ValueOf(v)
    switch rv.Kind() {
    case reflect.Ptr, reflect.UnsafePointer, reflect.Slice,
        reflect.Map, reflect.Chan, reflect.Func:
        if rv.IsNil() {
            panic("cgo: cannot use Handle for nil value")
        }

        k = rv.Pointer()
    default:
        // Wrap and turn a value parameter into a pointer.
        // This enables us to always store the passing object
        // as a pointer, and helps to identify which of whose
        // are initially pointers or values when Value is called.
        v = &wrap{v}
        k = reflect.ValueOf(v).Pointer()
    }

    ...
}
```

Note that the implementation above treats the values differently: For `reflect.Ptr`, `reflect.UnsafePointer`, `reflect.Slice`, `reflect.Map`, `reflect.Chan`, `reflect.Func` types, they are already pointers escaped to the heap, we can safely get the address from them. For the other kinds, we need to

turn them from a value to a pointer and also make sure they will always escape to the heap. That is the part:

```
        // Wrap and turn a value parameter into a pointer. This
        // enables us to always store the passing object as a
        // pointer, and helps to identify which of whose are
        // initially pointers or values when Value is called.
        v = &wrap{v}
        k = reflect.ValueOf(v).Pointer()
```

Now we have turned everything into an escaped value on the heap. The next thing we have to ask is: what if the two values are the same? That means the `v` passed to `cgo.NewHandle(v)` is the same object. Then we will get the same memory address in `k` at this point.

The easy case is, of course, if the address is not on the global map, then we do not have to think but return the address as the handle of the value:

```
func NewHandle(v interface{}) Handle {
    ...

    // v was escaped to the heap because of reflection. As Go do
    // not have a moving GC (and possibly lasts true for a long
    // future), it is safe to use its pointer address as the key
    // of the global map at this moment. The implementation must
    // be reconsidered if moving GC is introduced internally in
    // the runtime.
    actual, loaded := m.LoadOrStore(k, v)
    if !loaded {
        return Handle(k)
    }

    ...
}
```

Otherwise, we have to check the old value in the global map, if it is the same value, then we return the same address as expected:

```
func NewHandle(v interface{}) Handle {
    ...

    arv := reflect.ValueOf(actual)
    switch arv.Kind() {
    case reflect.Ptr, reflect.UnsafePointer, reflect.Slice,
        reflect.Map, reflect.Chan, reflect.Func:
        // The underlying object of the given Go value already have
        // its existing handle.
        if arv.Pointer() == k {
            return Handle(k)
```

```
        }

        // If the loaded pointer is inconsistent with the new
        // pointer, it means the address has been used for
        // different objects because of GC and its address is
        // reused for a new Go object, meaning that the Handle
        // does not call Delete explicitly when the old Go value
        // is not needed. Consider this as a misuse of a handle,
        // do panic.
        panic("cgo: misuse of a Handle")
    default:
        panic("cgo: Handle implementation has an internal bug")
    }
}
```

If the existing value shares the same address with the newly requested value, this must be a misuse of the Handle.

Since we have used the `wrap` struct to turn everything into the `reflect.Ptr` type, it is impossible to have other kinds of values to fetch from the global map. If that happens, it is an internal bug in the handle implementation.

When implementing the `Value()` method, we see why a `wrap` struct beneficial:

```
func (h Handle) Value() interface{} {
    v, ok := m.Load(uintptr(h))
    if !ok {
        panic("cgo: misuse of an invalid Handle")
    }
    if wv, ok := v.(*wrap); ok {
        return wv.v
    }
    return v
}
```

Because we can check when the stored object is a `*wrap` pointer, which means it was a value other than pointers. We return the value instead of the stored object.

Lastly, the `Delete` method becomes trivial:

```
func (h Handle) Delete() {
    _, ok := m.LoadAndDelete(uintptr(h))
    if !ok {
        panic("cgo: misuse of an invalid Handle")
    }
}
```

See a full implementation in golang.design/x/clipboard/internal/cgo.

The Accepted Approach

As one may have realized, the previous approach is much more complicated than expected and non-trivial: it relies on the foundation that runtime garbage collector is not a moving garbage collector, and an argument though interfaces will escape to the heap.

Although several other places in the internal runtime implementation rely on these facts, such as the channel implementation, it is still a little over-complicated than what we expected.

Notably, the previous `NewHandle` actually behaves to return a unique handle when the provided Go value refers to the same object. This is the core that brings the complexity of the implementation. However, we have another possibility: `NewHandle` always returns a different handle, and a Go value can have multiple handles.

Do we really need to Handle to be unique and keep it satisfy idempotence? After a short discussion with the Go team, we share the consensus that for the purpose of a Handle, it seems unnecessary to keep it unique for the following reasons:

1. The semantic of `NewHandle` is to return a *new* handle, instead of a unique handle;
2. The handle is nothing more than just an integer and guarantee it to be unique may prevent misuse of the handle, but it cannot always avoid the misuse until it is too late;
3. The complexity of the implementation.

Therefore, we need to rethink the original question: How to allocate a runtime-level unique ID?

In reality, the approach is more manageable: we only need to increase a number and never stop. This is the most commonly used approach for unique ID generation. For instance, in database applications, the unique id of a table row is always incremental; in Unix timestamp, the time is always incremental, etc.

If we use the same approach, what would be a possible concurrent-safe implementation? With `sync.Map` and atomic, we can produce code like this:

```
func NewHandle(v interface{}) Handle {
    h := atomic.AddUintptr(&handleIdx, 1)
    if h == 0 {
        panic("runtime/cgo: ran out of handle space")
    }

    handles.Store(h, v)
    return Handle(h)
}
```

```
var (
    handles   = sync.Map{} // map[Handle]interface{}
    handleIdx uintptr      // atomic
)
```

Whenever we want to allocate a new ID (`NewHandle`), one can increase the handle number `handleIdx` atomically, then the next allocation will always be guaranteed to have a larger number to use. With that allocated number, we can easily store it to a global map that persists all the Go values.

The remaining work becomes trivial. When we want to use the handle to retrieve the corresponding Go value back, we access the value map via the handle number:

```
func (h Handle) Value() interface{} {
    v, ok := handles.Load(uintptr(h))
    if !ok {
        panic("runtime/cgo: misuse of an invalid Handle")
    }
    return v
}
```

Further, if we are done with the handle, one can delete it from the value map:

```
func (h Handle) Delete() {
    _, ok := handles.LoadAndDelete(uintptr(h))
    if !ok {
        panic("runtime/cgo: misuse of an invalid Handle")
    }
}
```

In this implementation, we do not have to assume the runtime mechanism but just use the language. As long as the Go 1 compatibility keeps the promise `sync.Map` to work, there will be no need to rework the whole `Handle` design. Because of its simplicity, this is the accepted approach (see CL 295369[3]) by the Go team.

Aside from a future re-implementation of `sync.Map` that optimizes parallelism, the `Handle` will automatically benefit from it. Let us do a final benchmark that compares the previous method and the current approach:

```
func BenchmarkHandle(b *testing.B) {
    b.Run("non-concurrent", func(b *testing.B) {
        for i := 0; i < b.N; i++ {
            h := cgo.NewHandle(i)
            _ = h.Value()
            h.Delete()
        }
    })
    b.Run("concurrent", func(b *testing.B) {
```

```
        b.RunParallel(func(pb *testing.PB) {
            var v int
            for pb.Next() {
                h := cgo.NewHandle(v)
                _ = h.Value()
                h.Delete()
            }
        })
    })
}
```

```
name                      old time/op  new time/op  delta
Handle/non-concurrent-8   407ns ±1%    393ns ±2%   -3.51%  (p=0.000 n=8+9)
Handle/concurrent-8       768ns ±0%    759ns ±1%   -1.21%  (p=0.003 n=9+9)
```

Simpler, faster, why not?

Conclusion

This article discussed the newly introduced `runtime/cgo.Handle` facility coming in the Go 1.17 release that we contributed. The `Handle` facility enables us to pass Go values between Go and C back and forth without breaking the cgo pointer passing rules. After a short introduction to the usage of the feature, we first discussed a first attempt implementation based on the fact that the runtime garbage collector is not a moving GC and the escape behavior of `interface{}` arguments. After a few discussions of the ambiguity of the Handle semantics and the drawbacks in the previous implementation, we also introduced a straightforward and better-performed approach and demonstrated its performance.

As a real-world demonstration, we have been using the mentioned two approaches in two of our released packages for quite a long time: golang.design/x/clipboard and golang.design/x/hotkey [8] before in their `internal/cgo` package. We are looking forward to switching to the officially released `runtime/cgo` package in the Go 1.17 release.

For future work, one can foresee that a possible limitation in the accepted implementation is that the handle number may run out of the handle space very quickly in 32-bit or lower operating systems (similar to Year 2038 Problem. When we allocate 100 handles per second, the handle space can run out in 0xFFFFFFF / (24 * 60 * 60 * 100) = 31 days).

**If you are interested and think this is a serious issue, feel free to CC us when you send a CL, it would also be interesting for us to read your excellent approach.*

References

[1] Alex Dubov. 2020. runtime: provide centralized facility for managing (c)go pointer handles. The Go Project Issue Tracker. Feb 5. https://go.dev/issue/37033

[2] Changkun Ou. 2021. runtime/cgo: add Handle for managing (c)go pointers. The Go Project CL Tracker. Feb 21, 2021. https://go.dev/cl/294670

[3] Changkun Ou. 2021. runtime/cgo: add Handle for managing (c)go pointers. The Go Project CL Tracker. Feb 23, 2021. https://go.dev/cl/295369

[4] Ian Lance Taylor. 2015. cmd/cgo: specify rules for passing pointers between Go and C. The Go Project Issue Tracker. Aug 31. https://go.dev/issue/12416

[5] Ian Lance Taylor. 2015. Proposal: Rules for passing pointers between Go and C. The Go project design proposals. https://golang.org/design/12416-cgo-pointers

[6] Go Contributors. cgo. Mar 12, 2019. https://github.com/golang/go/wiki/cgo

[7] Changkun Ou. 2021. cross-platform clipboard package. The golang.design Initiative. Feb 25. https://github.com/golang-design/clipboard

[8] Changkun Ou. 2021. cross-platform hotkey package. The golang.design Initiative. Feb 27. https://github.com/golang-design/hotkey