Convert between slices of different types

Asked 10 years, 8 months ago Modified 4 months ago Viewed 72k times

65

I get a byte slice (`[]byte`) from a UDP socket and want to treat it as an integer slice (`[]int32`) without changing the underlying array, and vice versa. In C(++) I would just cast between pointer types; how would I do this in Go?

go

Share Improve this question

Follow

edited Dec 3, 2022 at 10:13

blackgreen ♦

29.8k ● 22 ● 103 ● 118

asked Aug 12, 2012 at 17:20

slartibartfast

4,330 ● 5 ● 30 ● 46

10 Answers

Sorted by:

Highest score (default)

60

As others have said, casting the pointer is considered bad form in Go. Here are examples of the proper Go way and the equivalent of the C array casting.

WARNING: all code untested.

The Right Way

In this example, we are using the `encoding/binary` package to convert each set of 4 bytes into an `int32`. This is better because we are specifying the endianness. We are also not using the `unsafe` package to break the type system.

```
import "encoding/binary"

const SIZEOF_INT32 = 4 // bytes

data := make([]int32, len(raw)/SIZEOF_INT32)
for i := range data {
    // assuming little endian
    data[i] = int32(binary.LittleEndian.Uint32(raw[i*SIZEOF_INT32:
(i+1)*SIZEOF_INT32]))
}
```

The Wrong Way (C array casting)

In this example, we are telling Go to ignore the type system. This is not a good idea because it may fail in another implementation of Go. It is assuming things not in the language specification. However, this one does not do a full copy. This code uses unsafe to access the

"SliceHeader" which is common in all slices. The header contains a pointer to the data (C array), the length, and the capacity. Instead of just converting the header to the new slice type, we first need to change the length and capacity since there are less elements if we treat the bytes as a new type.

```
import (
    "reflect"
    "unsafe"
)

const SIZEOF_INT32 = 4 // bytes

// Get the slice header
header := *(*reflect.SliceHeader)(unsafe.Pointer(&raw))

// The length and capacity of the slice are different.
header.Len /= SIZEOF_INT32
header.Cap /= SIZEOF_INT32

// Convert slice header to an []int32
data := *(*[]int32)(unsafe.Pointer(&header))
```

Share Improve this answer Follow edited Apr 13, 2013 at 4:50

answered Aug 13, 2012 at 1:55

Stephen Weinberg
50.6k ● 13 ● 133 ● 112

34 Of course the "right" way copies the data, the "wrong" way uses the data in place. – kristianp Apr 13, 2013 at 10:43

part of the memory safety of Go and their memory implementations, it isn't a guarantee of the language. although on the other hand i cant imagine that slices will ever be noncontiguous... – user1019517 Sep 28, 2014 at 3:42

The "right way" is not very performant when you have millions of items in the slice (i.e. pixels in an image) – Gillespie Oct 14, 2022 at 14:46

9

You do what you do in C, with one exception - Go does not allow to convert from one pointer type to another. Well, it does, but you must use unsafe.Pointer to tell compiler that you are aware that all rules are broken and you know what you are doing. Here is an example:

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    b := []byte{1, 0, 0, 0, 2, 0, 0, 0}

    // step by step
    pb := &b[0]         // to pointer to the first byte of b
    up := unsafe.Pointer(pb)    // to *special* unsafe.Pointer, it can be
converted to any pointer
    pi := (*[2]uint32)(up)      // to pointer to the first uint32 of array of 2
uint32s
```

```
    i := (*pi)[:]           // creates slice to our array of 2 uint32s (optional
step)
    fmt.Printf("b=%v i=%v\n", b, i)

    // all in one go
    p := (*[2]uint32)(unsafe.Pointer(&b[0]))
    fmt.Printf("b=%v p=%v\n", b, p)
}
```

Obviously, you should be careful about using "unsafe" package, because Go compiler is not holding your hand anymore - for example, you could write `pi := (*[3]uint32)(up)` here and compiler wouldn't complain, but you would be in trouble.

Also, as other people pointed already, bytes of uint32 might be layout differently on different computers, so you should not assume these are layout as you need them to be.

So safest approach would be to read your array of bytes one by one and make whatever you need out of them.

Alex

Share Improve this answer Follow

edited Aug 13, 2012 at 0:24

poolie
9,230 ● 1 ● 47 ● 73

answered Aug 13, 2012 at 0:10

alex
2,158 ● 16 ● 13

This converts it to a fixed size array. If you don't know the number of int32s at compile time, this will not work. – Stephen Weinberg Aug 13, 2012 at 3:30

1 Why not? I can say that array is as big as I want it to be. – alex Aug 13, 2012 at 11:45

Yes you can. But you can only do so at compile time. You can not convert it to an `[n]uint32` where `n` is variable. – Stephen Weinberg Aug 13, 2012 at 15:54

1 Everything you say is correct. In your example, you convert the `[]byte` to a `[2]uint32`. That is an array. I was trying to point out that if you did not know how many int32's were being converted at compile time, your method of using an array would not work. – Stephen Weinberg Aug 14, 2012 at 0:25

1 Sure you might not know how big you data is, but you can always say it is less then some const value. For example, you can convert it into [1<<20]uint32. My proposal does not change in any way if final array size changes, but it will give you access to as much data as you need - you could even create slice of size n to it to make it what you want. – alex Aug 14, 2012 at 0:41

7

The short answer is you can't. Go wont let you cast a slice of one type to a slice of another type. You will have loop through the array and create another array of the type you want while casting each item in the array. This is generally regarded as a good thing since typesafety is an important feature of go.

Share Improve this answer Follow

answered Aug 12, 2012 at 19:33

Jeremy Wall
23.5k ● 5 ● 55 ● 73

4 It seems inconsistent to allow `int8(int16(a))` (which crashes at runtime if `a > 255`, but forbid `[]int8([]int16(a))`, which is equally (un)safe (and would be perfectly safe in a case like `type myint int; []myint([]int{1})`). Oh, well. – misterbee Jul 25, 2013 at 4:26

1 You may think so but those are two different cases in Go's type system and can't really be conflated. Go's type system is not theoretically perfect but instead meant to increase developer productivity. – Jeremy Wall Jul 25, 2013 at 15:33

8 Well, in this case the type system doesn't increase developer productivity. I'm not suggesting that Go should change to support this case at the expense of anything else, just pointing out that it's a design wart. I'm pretty confident that it's an inconsistency. – misterbee Jul 25, 2013 at 18:59

6

Since Go 1.17, there is a simpler way to do this using the `unsafe` package.

```
import (
    "unsafe"
)

const SIZEOF_INT32 = unsafe.Sizeof(int32(0)) // 4 bytes

func main() {
    var bs []byte

    // Do stuff with `bs`. Maybe do some checks ensuring that len(bs) %
SIZEOF_INT32 == 0

    data := unsafe.Slice((*int32)(unsafe.Pointer(&bs[0])), len(bs)/SIZEOF_INT32)

    // A more verbose alternative requiring `import "reflect"`
    // data := unsafe.Slice((*int32)(unsafe.Pointer((*reflect.SliceHeader)
(unsafe.Pointer(&bs)).Data)), len(bs)/SIZEOF_INT32)
}
```

Share Improve this answer Follow

answered Aug 25, 2021 at 21:17

YenForYang
2,788 ● 22 ● 21

3

Go 1.17 and beyond

Go 1.17 introduced the `unsafe.Slice` function, which does exactly this.

Converting a `[]byte` to a `[]int32`:

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    theBytes := []byte{
        0x33, 0x44, 0x55, 0x66,
        0x11, 0x22, 0x33, 0x44,
```

```
        0x77, 0x66, 0x55, 0x44,
    }

    numInts := uintptr(len(theBytes)) * unsafe.Sizeof(theBytes[0]) /
unsafe.Sizeof(int32(0))
    theInts := unsafe.Slice((*int32)(unsafe.Pointer(&theBytes[0])), numInts)

    for _, n := range theInts {
        fmt.Printf("%04x\n", n)
    }
}
```

Playground.

Share Improve this answer Follow

answered Oct 22, 2021 at 11:29

rodrigocfd
5,746 ● 5 ● 31 ● 64

2

I had the size unknown problem and tweaked the previous unsafe method with the following code. given a byte slice b ...

```
int32 slice is (*(*[]int)(Pointer(&b)))[:len(b)/4]
```

The array to slice example may be given a fictional large constant and the slice bounds used in the same way since no array is allocated.

Share Improve this answer Follow edited Jul 30, 2013 at 10:10

answered Jul 29, 2013 at 18:32

Tony Wilson
21 ● 2

1

You can do it with the "unsafe" package

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var b [8]byte = [8]byte{1, 2, 3, 4, 5, 6, 7, 8}
    var s *[4]uint16 = (*[4]uint16)(unsafe.Pointer(&b))
    var i *[2]uint32 = (*[2]uint32)(unsafe.Pointer(&b))
    var l *uint64 = (*uint64)(unsafe.Pointer(&b))

    fmt.Println(b)
    fmt.Printf("%04x, %04x, %04x, %04x\n", s[0], s[1], s[2], s[3])
    fmt.Printf("%08x, %08x\n", i[0], i[1])
    fmt.Printf("%016x\n", *l)
}

/*
 * example run:
 * $ go run /tmp/test.go
```

```
 * [1 2 3 4 5 6 7 8]
 * 0201, 0403, 0605, 0807
 * 04030201, 08070605
 * 0807060504030201
 */
```

Share Improve this answer Follow

answered Oct 11, 2016 at 17:57

teknoraver
896 ● 8 ● 6

1

Perhaps it was not available when the earlier answers were given, but it would seem that the `binary.Read` method would be a better answer than "the right way" given above.

This method allows you to read binary data from a reader directly into the value or buffer of your desired type. You can do this by creating a reader over your byte array buffer. Or, if you have control of the code that is giving you the byte array, you can replace it to read directly into your buffer without the need for the interim byte array.

See https://golang.org/pkg/encoding/binary/#Read for the documentation and a nice little example.

Share Improve this answer Follow

answered Mar 6, 2019 at 16:50

Steven W. Klassen
1,361 ● 12 ● 26

0

http://play.golang.org/p/w1m5Cs-ecz

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := []interface{}{"foo", "bar", "baz"}
    b := make([]string, len(s))
    for i, v := range s {
        b[i] = v.(string)
    }
    fmt.Println(strings.Join(b, ", "))
}
```

Share Improve this answer Follow

answered May 19, 2014 at 10:30

KIM Taegyoon
1,907 ● 21 ● 18

0

```go
func crackU32s2Bytes(us []uint32) []byte {
    var bs []byte
    var ptrBs = (*reflect.SliceHeader)(unsafe.Pointer(&bs))
```

```
    var ptrUs = (*reflect.SliceHeader)(unsafe.Pointer(&us))
    ptrBs.Data = ptrUs.Data
    ptrBs.Len = ptrUs.Len*4
    ptrBs.Cap = ptrBs.Len
    return bs
}

func crackBytes2U32s(bs []byte) []uint32 {
    var us []uint32
    var ptrBs = (*reflect.SliceHeader)(unsafe.Pointer(&bs))
    var ptrUs = (*reflect.SliceHeader)(unsafe.Pointer(&us))
    ptrUs.Data = ptrBs.Data
    ptrUs.Len = ptrBs.Len/4
    ptrUs.Cap = ptrUs.Len
    return us
}
```

Share Improve this answer Follow

answered Mar 28, 2022 at 12:24

user3222762

123 ● 1 ● 1 ● 5

Your answer could be improved with additional supporting information. Please edit to add further details, such as citations or documentation, so that others can confirm that your answer is correct. You can find more information on how to write good answers in the help center. – Community Bot Mar 28, 2022 at 14:49