图解Go之内存对齐

苗蕾（newbmiao）2020.4.2

https://github.com/talk-go/night/issues/588

提纲

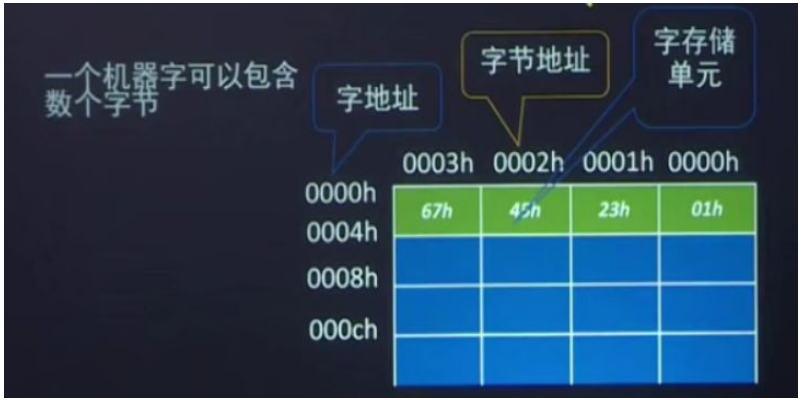
- 了解内存对齐的收益
- 为什么要对齐
- 怎么对齐:
  - 数据结构对齐
  - 内存地址对齐
- 64位字的安全访问保证(32位平台)

了解内存对齐的收益

- 提高代码平台兼容性
- 优化数据对内存的使用
- 避免一些内存不对齐带来的坑
- 有助于一些源码的阅读

为什么要对齐

位 bit	计算机内部数据存储的最小单位
字节 byte	计算机数据处理的基本单位
机器字 machine word	计算机用来一次性处理事务的一个固定长度



为什么要对齐

1.平台原因(移植原因):

不是所有的硬件平台都能访问任意地址上的任意数据的;某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2.性能原因:

数据结构应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作**两次内存访问**;而对齐的内存访问仅需要**一次访问**。

数据结构对齐 大小保证(size guarantee)

type	size in bytes
byte, uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64, float64, complex64	8
complex128	16
struct{}, [0]T{}	0

数据结构对齐 对齐保证(align guarantee)

type	alignment guarantee
bool, byte, uint8, int8	1
uint16, int16	2
uint32, int32	4
float32, complex64	4
arrays	由其元素(element)类型决定
structs	由其字段(field)类型决定
other types	一个机器字(machine word)的大小

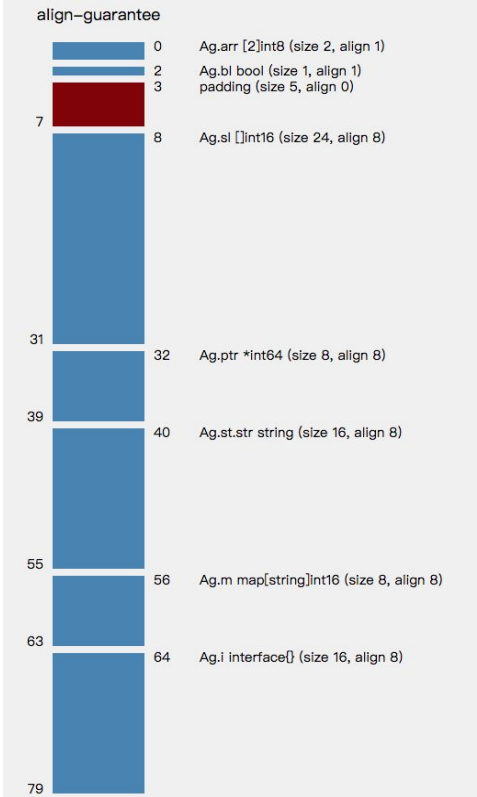
数据结构对齐工具

```
# layout
go get github.com/ajstarks/svgo/structlayout-svg
go get -u honnef.co/go/tools
go install honnef.co/go/tools/cmd/structlayout
go install honnef.co/go/tools/cmd/structlayout-pretty

# optmize
go install honnef.co/go/tools/cmd/structlayout-optimize
```

数据结构对齐 举个栗子

```go
type Ag struct {
    arr [2]int8  // 2
    bl  bool     // 1 padding 5
    sl  []int16  // 24
    ptr *int64   // 8
    st  struct { // 16
        str string
    }
    m map[string]int16
    i interface{}
}
```




```
structlayout -json github.com/NewbMiao/Dig101-Go Ag|structlayout-svg -t "align-guarantee" > ag.svg
```

数据结构对齐 几个底层数据结构

```go
// reflect/value.go
type StringHeader struct {
    Data uintptr
    Len  int
}

type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

```go
// runtime/map.go
type hmap struct {
    count      int
    flags      uint8
    B          uint8
    noverflow  uint16
    hash0      uint32
    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr
    extra      *mapextra
}
```

```go
// runtime/runtime2.go
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

数据结构对齐 举个特栗子:final zero field

```
type T1 struct {
    a struct{}
    x int64
}
type T2 struct {
    x int64
    a struct{}
}
a1 := T1{}
a2 := T2{}
fmt.Printf("zero size struct{} of T1 size: %d; T2 (as final field) size: %d",
    unsafe.Sizeof(a1),// 8
    unsafe.Sizeof(a2))// 64位: 16; 32位: 12
```

数据结构对齐 重排优化(粗暴方式-按对齐值的递减来重排成员)

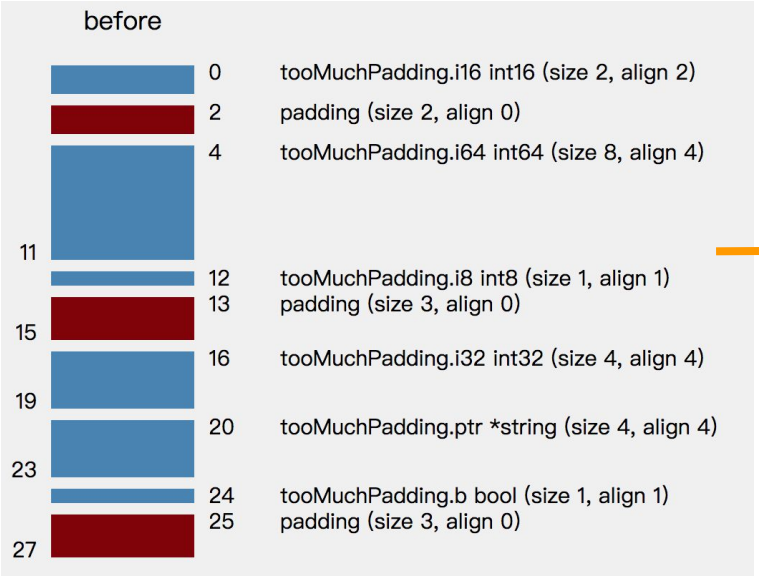
```
type tooMuchPadding struct {
    i16 int16
    i64 int64
    i8  int8
    i32 int32
    ptr *string
    b   bool
}
```

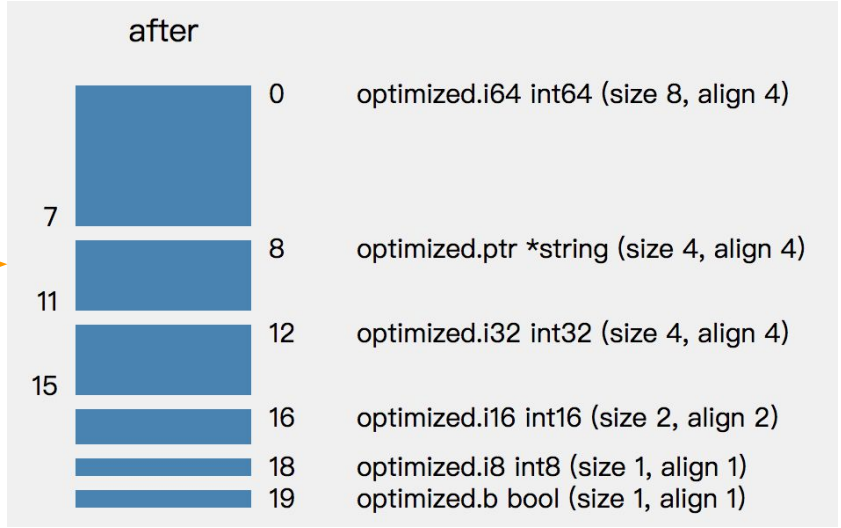
-40%

```
tooMuchPadding.i64 int64: 0-8 (size 8, align 8)
tooMuchPadding.ptr *string: 8-16 (size 8, align 8)
tooMuchPadding.i32 int32: 16-20 (size 4, align 4)
tooMuchPadding.i16 int16: 20-22 (size 2, align 2)
tooMuchPadding.i8 int8: 22-23 (size 1, align 1)
tooMuchPadding.b bool: 23-24 (size 1, align 1)
```

```
structlayout -json github.com/NewbMiao/Dig101-Go tooMuchPadding|structlayout-optimize -r
```

数据结构对齐 重排优化


before
0
tooMuchPadding.i16 int16 (size 2, align 2)
2
padding (size 2, align 0)
4
tooMuchPadding.i64 int64 (size 8, align 4)
11
12
tooMuchPadding.i8 int8 (size 1, align 1)
13
padding (size 3, align 0)
15
16
tooMuchPadding.i32 int32 (size 4, align 4)
19
20
tooMuchPadding.ptr *string (size 4, align 4)
23
24
tooMuchPadding.b bool (size 1, align 1)
25
padding (size 3, align 0)
27


after
0
optimized.i64 int64 (size 8, align 4)
7
8
optimized.ptr *string (size 4, align 4)
11
12
optimized.i32 int32 (size 4, align 4)
15
16
optimized.i16 int16 (size 2, align 2)
18
optimized.i8 int8 (size 1, align 1)
19
optimized.b bool (size 1, align 1)

数据结构对齐 内存对齐检测

```
golangci-lint run --disable-all --enable maligned   struct_align_demo.go

struct_align_demo.go:17:21: struct of size 40 bytes could be of size 24 bytes (maligned)
type tooMuchPadding struct {
```

github.com/NewbMiao/Dig101-Go/struct_align_demo.go

内存地址对齐

计算机结构可能会要求内存地址 进行对齐;也就是说，一个变量的**地址**是一个因子的倍数，也就是该变量的类型是对齐值。

函数**Alignof**接受一个表示任何 类型变量的表达式作 为参数，并以字 节为单位返回变量(类型)的对齐值。对于变量x:

```
uintptr(unsafe.Pointer(&x)) % unsafe.Alignof(x) == 0
```

https://golang.org/ref/spec#Package_unsafe

内存地址对齐 举个🌰

```go
type WaitGroup struct {
  noCopy noCopy
  state1 [3]uint32
}

func (wg *WaitGroup) state() (statep *uint64, semap *uint32) {
  // 判定地址是否8位对齐
  if uintptr(unsafe.Pointer(&wg.state1))%8 == 0 {
    // 前8bytes做uint64指针statep，后4bytes做sema
    return (*uint64)(unsafe.Pointer(&wg.state1)), &wg.state1[2]
  } else { // 否则相反
    return (*uint64)(unsafe.Pointer(&wg.state1[1])), &wg.state1[0]
  }
}
```

为什么是[3]uint32, 不是[12]byte

64位字的安全访问保证(32位系统)

在x86-32上，64位函数使用Pentium MMX之前不存在的指令。

在非Linux ARM上，64位函数使用ARMv6k内核之前不可用的指令。

在ARM，x86-32和32位MIPS上，调用方有责任安排对原子访问的64位字的64位对齐。变量或分配的结构、数组或切片中的第一个字（word）可以依赖当做是64位对齐的。

https://golang.org/pkg/sync/atomic/#pkg-note-BUG

64位字的安全访问保证 Why？

这是因为int64在bool之后未对齐。

它是32位对齐的，但不是64位对齐的，因为我们使用的是32位系统，

因此实际上只是两个32位值并排在一起。

https://github.com/golang/go/issues/6404#issuecomment-66085602

```
type WillPanic struct {
  init      bool
  uncounted int64
}
```

64位字的安全访问保证 How？

变量或**已分配**的结构体、数组或切片中的第一个字（word）可以依赖当做是64位对齐的。

The first word in a variable or in an allocated struct, array, or slice can be relied upon to be 64-bit aligned.

```
// GOARCH=386 go run types/struct/struct.go
var c0 int64
fmt.Println("64位字本身: ",
    atomic.AddInt64(&c0, 1))
c1 := [5]int64{}
fmt.Println("64位字数组、切片:",
    atomic.AddInt64(&c1[:][0], 1))
```

已分配：new 或者make

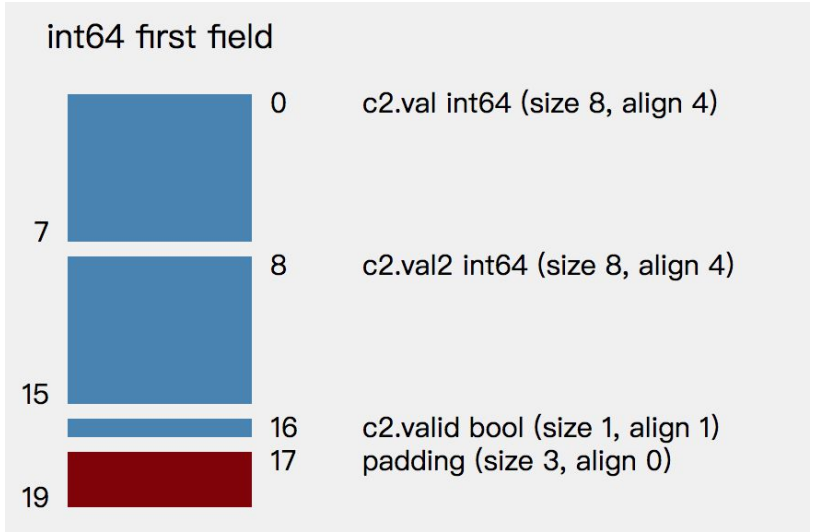
```
go run -gcflags="-m=2" types/struct/struct.go

types/struct/struct.go:102:6: moved to heap: c0
types/struct/struct.go:106:2: moved to heap: c1
types/struct/struct.go:110:2: moved to heap: c2
types/struct/struct.go:123:2: moved to heap: c3
types/struct/struct.go:129:2: moved to heap: c4
types/struct/struct.go:145:17: []int64 literal escapes to heap
types/struct/struct.go:163:13: new(int64) escapes to heap

GOSSAFUNC=safeAtomicAccess64bitWordOn32bitArch go run types/struct/struct.go
```

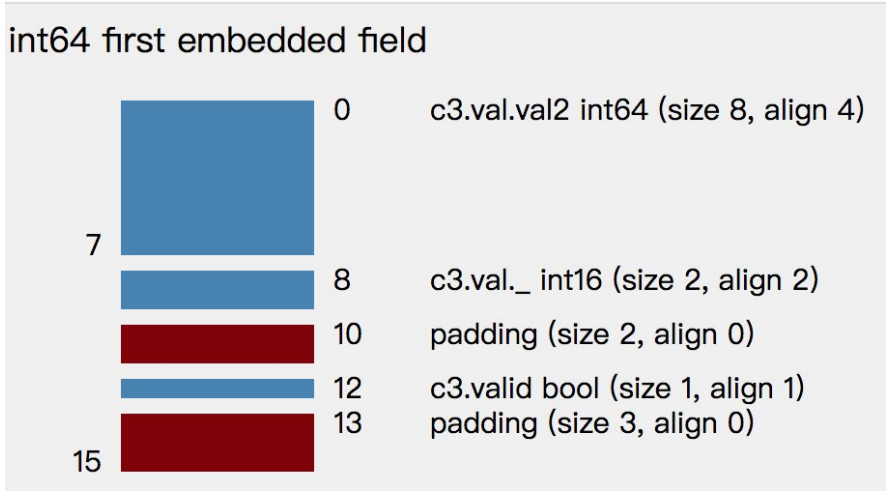
64位字的安全访问保证 How？

```
c2 := struct {
    val   int64 // pos 0
    val2  int64 // pos 8
    valid bool  // pos 16
}{}
fmt.Println("结构体首字段为对齐的64位字及相邻的64位字:",
    atomic.AddInt64(&c2.val, 1),
    atomic.AddInt64(&c2.val2, 1))
```



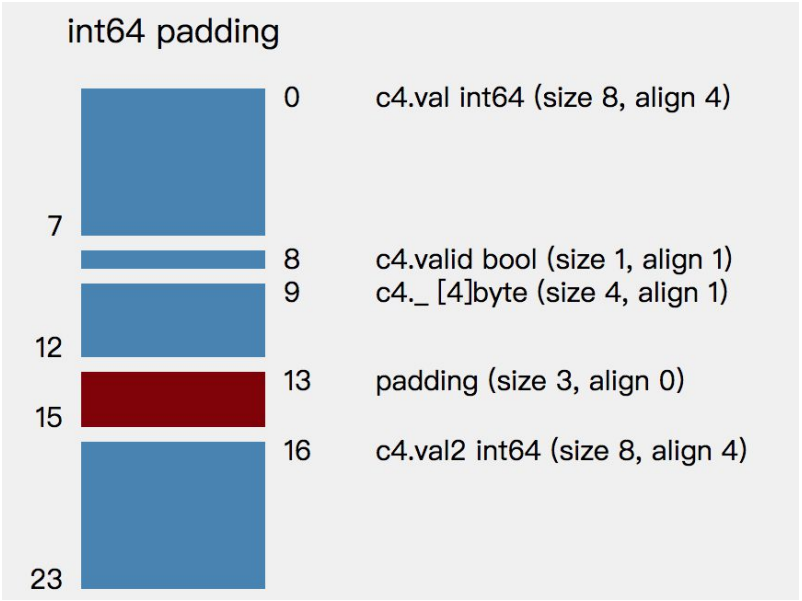
64位字的安全访问保证 How？

```
type T struct {
    val2 int64
    _    int16
}
c3 := struct {
    val   T
    valid bool
}{}
fmt.Println("结构体中首字段为嵌套结构体，且其首元素为64位字:",
    atomic.AddInt64(&c3.val.val2, 1))
```



64位字的安全访问保证 How？

```
c4 := struct {
    val   int64    // pos 0
    valid bool     // pos 8
    // 或者 _ uint32
    // 使32位系统上多填充 4bytes
    _     [4]byte // pos 9
    val2  int64    // pos 16
}{}
fmt.Println("结构体增加填充使对齐的64位字:",
    atomic.AddInt64(&c4.val2, 1))
```

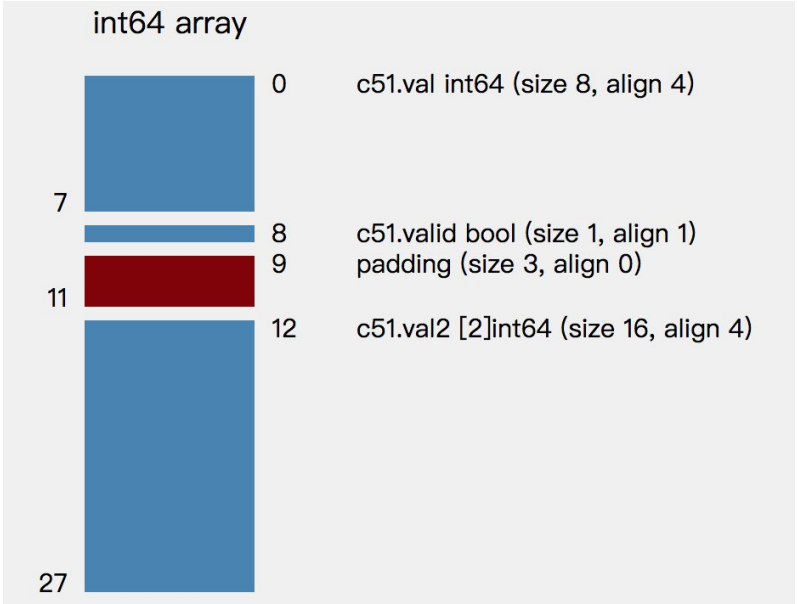


64位字的安全访问保证 How？

```
c5 := struct {
    val   int64
    valid bool
    val2 []int64
}{val2: []int64{0}}
fmt.Println("结构体中64位字切片:",
    atomic.AddInt64(&c5.val2[0], 1))
```

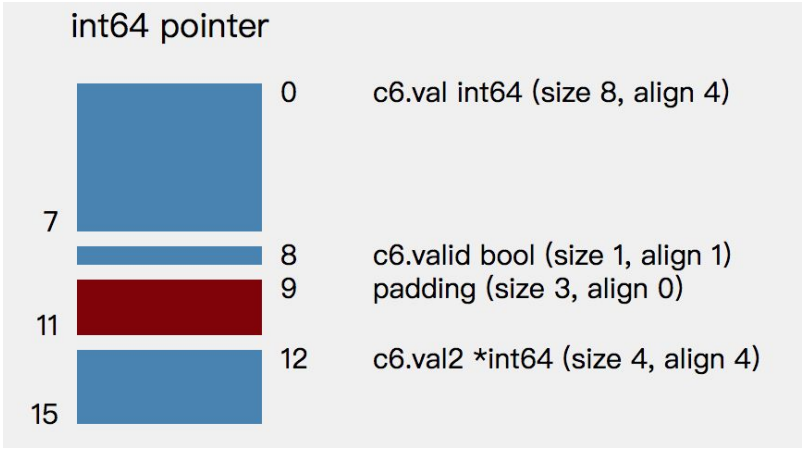
64位字的安全访问保证 How？

```
c51 := struct {
  val   int64
  valid bool
  val2  [3]int64
}{val2: [3]int64{0}}
// will panic
atomic.AddInt64(&c51.val2[0], 1)
```



64位字的安全访问保证 How？

```
c6 := struct {
    val   int64
    valid bool
    val2  *int64
}{val2: new(int64)}
fmt.Println("结构体中64位字指针:",
    atomic.AddInt64(c6.val2, 1))
```



一些源码中的🌰

```
type p struct {
    ...
    _ uint32 // Alignment for atomic fields below
    timer0When uint64
    ...
    pad cpu.CacheLinePad
}
```

GMP中的管理groutine本地队列的上下文p中，记录计时器运行时长的uint64，需要保证32位系统上也是8byte对齐（原子操作）

一些源码中的🌰

```
type mheap struct {
    ...
    _ uint32 // ensure 64-bit alignment of central
    central [numSpanClasses]struct {
        mcentral mcentral
        pad      [cpu.CacheLinePadSize - unsafe.Sizeof(mcentral{})%cpu.CacheLinePadSize]byte
    }
    ...
}
```

堆对象分配的mheap中，管理全局cache的中心缓存列表central，分配或释放需要加互斥锁
另外为了**不同列表间互斥锁**不会伪共享，增加了cacheLinePadding

cacheLine 参考：https://appliedgo.net/concurrencyslower/

64位字的安全访问保证 Bug！

如果包含首个64位字的结构体是12byte大小时，不一定能保证64未对齐

这**是tinyalloc**分配小对象时没有做对齐保证

```
func mallocgc(size uintptr, typ *_type, needzero bool) unsafe.Pointer {
    ...
    c := gomcache()
    var x unsafe.Pointer
    noscan := typ == nil || typ.ptrdata == 0
    if size <= maxSmallSize {
        if noscan && size < maxTinySize {
            // Tiny allocator.
            off := c.tinyoffset
            // Align tiny pointer for required (conservative) alignment.
            if size&7 == 0 {
                off = alignUp(off, 8)
            } else if size&3 == 0 { // <= 12bytes 1100 & 0011 == 0
                off = alignUp(off, 4)
            } else if size&1 == 0 {
                off = alignUp(off, 2)
            }
            if off+size <= maxTinySize && c.tiny != 0 {
                // The object fits into existing tiny block.
                x = unsafe.Pointer(c.tiny + off)
                c.tinyoffset = off + size
                c.local_tinyallocs++
                mp.mallocing = 0
                releasem(mp)
                return x
            }
    ...
```

https://github.com/golang/go/issues/37262#issuecomment-587576192

64位字的安全访问保证 改为加锁！

```
c := struct{
    val int16
    val2 int64
}{}
var mu sync.Mutex
mu.Lock()
c.val2 += 1
mu.Unlock()
```

总结

- 内存对齐是为了cpu更**高效**访问内存中数据
- 结构体对齐依赖类型的**大小保证**和**对齐保证**
- 地址对齐保证是:如果类型 t 的对齐保证是 n，那么类型 t 的每个值的**地址**在运行时必须是 n 的倍数。
- struct内字段如果**填充**过多，可以尝试**重排**，使字段排列更紧密，减少内存浪费
- 零大小字段要避免作为struct最后一个字段，会有内存浪费
- 32位系统上对64位字的原子访问要保证其是8bytes对齐的;当然如果不必要的话，还是用加锁(`mutex`)的方式更清晰简单

Thanks

了解更多

- Go101- Memory Layouts
- Dig101-Go之聊聊struct的内存对齐
- dominikh/go-tools

Go 夜读

菜鸟Miao