底层编程

把C学长叫过来！😊

unsafe

`package unsafe`，底层编程的危险派对~

unsafe.Sizeof

求变量/类型在内存中占用的字节数
e.g.

```
var i int
    var f float64
    var s string = "hello"

    fmt.Println("int size:", unsafe.Sizeof(i))      // 通常8字节(64位系统)
    fmt.Println("float64 size:", unsafe.Sizeof(f))  // 8字节
    fmt.Println("string size:", unsafe.Sizeof(s))   // 16字节(64位系统)

    type Person struct {
            Name string
            Age  int
    }
    fmt.Println("struct size:", unsafe.Sizeof(Person{}))  // 24字节(16+8)
```

unsafe.Alignof & unsafe.Offsetof

内存地址对齐?

内存对齐是指数据在内存中的存储起始地址必须是某个值（通常是2、4、8等2的幂次方）的整数倍
e.g.

```
var x struct {
    a bool // 1字节
    b int16 // 2字节
    c [] int // 如下图所示
}
```

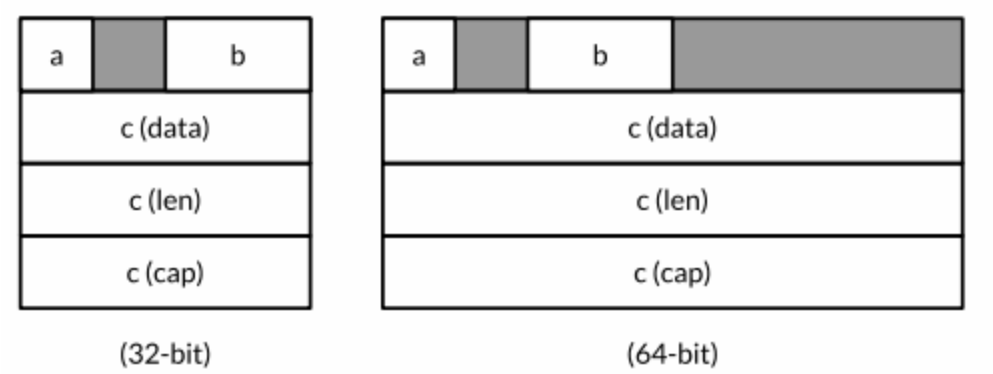

Figure 13.1. Holes in a struct.

（32/64位系统不一样）

对齐值通常是2的幂次方，结构体的对齐值由其字段中最大的对齐值决定，对齐后方便后续内存访问

为了对齐可能会出现空洞（图中灰色部分），但这也算在消耗的内存中

`Alignof()` 返回对应参数的类型需要对齐的倍数

常情况下布尔和数字类型需要对齐到它们本身的大小（最多8个字节），其它的类型对齐到机器字大小

e.g.

类型	大小	对齐值(Alignof返回值)	解释
`bool`	1	1	可以放在任何地址（1的倍数）
`int8`	1	1	同bool
`int16`	2	2	起始地址必须是2的倍数
`int32`	4	4	起始地址必须是4的倍数
`int64`	8	8	起始地址必须是8的倍数
`float64`	8	8	同int64

类型	大小	对齐值(Alignof返回值)	解释
string	16	8	虽然总大小16字节，但按机器字(8字节)对齐
[]int	24	8	切片描述符按机器字对齐

Offsetof()
必须是结构体的某个字段 x.f ， 返回 f 字段相对于 x 起始地址的偏移量

e.g.

```
func unsafeEx2() {
        type P struct {
                a bool // 1字节，对齐到1字节
                b int16 // 2字节，对齐到2字节
                c []int // 3x8字节（64位系统），对齐到机器字长8字节
        }
        x := P{}
        fmt.Println("Sizeof:", unsafe.Sizeof(x.a), unsafe.Sizeof(x.b), unsafe.Sizeof(x.c))
        fmt.Println("Alignof:", unsafe.Alignof(x.a), unsafe.Alignof(x.b), unsafe.Alignof(x.c))
        fmt.Println("Offsetof:", unsafe.Offsetof(x.a), unsafe.Offsetof(x.b), unsafe.Offsetof(x.c
}
```

结果（64位系统）：

```
Sizeof: 1 2 24
Alignof: 1 2 8
Offsetof: 0 2 8 # 如上面的图所示
```

unsafe.Pointer

指针？底层不老铁？！ 😊 👍
*T 指向T类型的指针
那 unsafe.Pointer 你来干嘛的

类似C中的 void* 类型的指针，可以包含任意类型变量的地址

*T 和 unsafe.Pointer 互转

e.g.

```
// *float64 -> *uint64
func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }

fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

说实话🤔

有点神经了😋

获取结构体字段

```
var x struct {
    a bool
    b int16
    c []int
}

// 和 pb := &x.b 等价
pb := (*int16)(unsafe.Pointer(
    uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42
fmt.Println(x.b) // "42"
```

说实话🤔

更神经了😋 😋

uintptr闭嘴

uintptr 可以存储一个 和当前指针相同的数字值，并不是一个指针

因为Go的GC垃圾回收可能会导致变量的内存地址发生变化，指针有东西指，所以知道，会跟着变；但是 uintptr 就是一个定死的数，它不会跟着变，所以最好不要引入一个 uintptr 类型的中间变量并把它作为 unsafe.Pointer() 的参数！

e.g.

```
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b) // uintptr类型，就是一个普通的数，比如说
pb := (*int16)(unsafe.Pointer(tmp)) // 可能内存地址已经变了（比如原来2位置的变到1位置了），但uintpt
*pb = 42 // 把现在在2位置其他的东西给搞掉了，那不是乱搞了吗
```

再比如

```
pT := uintptr(unsafe.Pointer(new(T)))
// new()完之后，没有指针引用它（没有，垃圾），垃圾收集器可能立马回收其空间
// 那你还unsafe.Pointer()个甚啊，皇帝都没了你去哪上贡
```

通过cgo调用C代码

Go里面跑C，哈基Go，你这家伙，为了击败C学长不惜cos成C吗？😋 😋

序言 & 序言注释

`import "C"` ——序言
真的有个 `C` 库？骗你的，这是cos服

序言注释，在 `import "C"` 前面的 `/* */` 注释中编写C代码，然后就可以跑了

e.g. `C.hello()`

```go
package main

/*
#include <stdio.h>

void hello() {
    printf("hello, world!\n");
}
*/
import "C"

func main() {
    C.hello() // 调用C函数
}
```

注意，Go比较傲娇（为了成为标准的轮椅的必要牺牲）
注释用 `//` 不行，报错
注释和 `import "C"` 之间有空行不行，报错
C代码写错了？不行，库库报错

基础用法

基本类型转换

in C	in Go
char	C.char
int	C.int
unsigned int	C.uint
long	C.long
double	C.double
char*	*C.char

字符串转换

```
package main

/*
#include <string.h>
#include <stdlib.h>
*/
import "C"
import "unsafe"

func main() {
    // Go str 转 C
        goStr := "Hello, C!"
        cStr := C.CString(goStr)
        defer C.free(unsafe.Pointer(cStr)) // 必须手动释放内存

        // CString方法
        length := C.strlen(cStr)
        println("String length:", length)

        // C str 转 Go
        goStrBack := C.GoString(cStr)
        println(goStrBack)
}
```

本来是C里面的 `strlen()` 函数，import进来之后直接 `C.strlen()` 就OK

结构体

让Go小兄弟看看 `typedef struct{}`

```go
package main

/*
typedef struct {
    int x;
    int y;
} Point;

int sum(Point p) {
    return p.x + p.y;
}
*/
import "C"
import "fmt"

func main() {
    p := C.Point{x: 10, y: 20}
    result := C.sum(p)
    fmt.Println("Sum:", result) // 输出: Sum: 30
}
```

#cgo 链接C库

#cgo 指令语法

```go
/*
#cgo [GOOS/GOARCH...] [CFLAGS/LDFLAGS...] 编译选项
*/
```

e.g.

```go
/*
#cgo CFLAGS: -I/usr/local/include
#cgo LDFLAGS: -L/usr/local/lib -lfoo
#cgo windows LDFLAGS: -lbar
*/
```

看不懂思密达😋
（上完编译原理再说吧）

圣经的评价

ch12 和 ch13是不是很底层很抽象？那圣经是如何评价的呢？

《虽然反射提供的API远多于我们讲到的，我们前面的例子主要是给出了一个方向，通过反射可以实现哪些功能。反射是一个强大并富有表达力的工具，但是它应该被小心地使用》

《我们在前一章结尾的时候，我们警告要谨慎使用reflect包。那些警告同样适用于本章的unsafe包》

《大多数Go程序员可能永远不会需要直接使用unsafe》

《现在，赶紧将最后两章抛入脑后吧。编写一些实实在在的应用是真理。请远离reflect和unsafe包，除非你确实需要它们》

😋 👍