# unsafe 非类型安全操作

unsafe库徘徊在“类型安全”边缘，由于它们绕过了 Golang 的内存安全原则，一般被认为使用该库是不安全的。但是，在许多情况下，unsafe库的作用又是不可替代的，灵活地使用它们可以实现对内存的直接读写操作。在reflect库、syscall库以及其他许多需要操作内存的开源项目中都有对它的引用。

unsafe库源码极少，只有两个类型的定义和三个方法的声明。

## 常见函数

|  |  |
| --- | --- |
| **函数签名** | **说明** |
| func Sizeof(v ArbitraryType) uintptr | 返回变量 v 占用的内存空间的字节数，该字节数不是按照变量 v 实际占用的内存计算，而是按照 v 的“ top level ”内存计算。比如，在 64 位系统中，如果变量 v 是 int 类型，会返回 16，因为 v 的“ top level ”内存就是它的值使用的内存；如果变量 v 是 string 类型，会返回 16，因为 v 的“ top level ”内存不是存放着实际的字符串，而是该字符串的地址；如果变量 v 是 slice 类型，会返回 24，这是因为 slice 的描述符就占了 24 个字节 |
| func Offsetof(v ArbitraryType) uintptr | 返回由 v 所指示的某结构体中的字段在该结构体中的位置偏移字节数，注意，v 的表达方式必须是“ struct.filed ”形式 |
| func Alignof(v ArbitraryType) uintptr | 相当于 reflect.TypeOf(x).Align() |

## Arbitrary 类型与 Pointer 类型

type Pointer \*ArbitraryType

* 官方导出类型`ArbitraryType`只是出于完善文档的考虑，在其他的库和任何项目中都没有使用价值，除非程序员故意使用它。
* 类型`Pointer`比较重要，它是实现定位欲读写的内存的基础。官方文档对该类型有四个重要描述：

1. 任何类型的指针都可以被转化为 Pointer

2. Pointer 可以被转化为任何类型的指针

3. uintptr 可以被转化为 Pointer

4. Pointer 可以被转化为 uintptr

针对第一二点，我们可以实现类型转换：

package main  
  
import (  
 "fmt"  
 "unsafe"  
)  
  
func main() {  
 var f float64 = 5.5  
 ptr := (\*int64)(unsafe.Pointer(&f))  
 fmt.Println(\*ptr) //输出：4617878467915022336  
 fmt.Println(int64(f)) //输出：5  
}

可见，由于通过不同的数据类型在内存中的数据布局不一样，并不能直接转换，否则容易达到非预期的结果

# 关于unsafe包的详细解析

|  |  |
| --- | --- |
| **函数** | **说明** |
| func Sizeof(x ArbitraryType) uintptr | 返回的大小只包括数据结构中固定的部分 |
| func Alignof(x ArbitraryType) uintptr | 返回对应参数的类型需要对齐的倍数 |
| func Offsetof(x ArbitraryType) uintptr | 参数必须是一个字段 x.f，然后返回 f 字段相对于 x 起始地址的偏移量，包括可能的空洞 |

无符号的整数类型uintptr，没有指定具体的bit大小但是足以容纳指针。uintptr类型只有在底层编程时才需要，特别是Go语言和C语言函数库或操作系统接口相交互的地方

## **1. unsafe.Sizeof, Alignof 和 Offsetof**

unsafe.Sizeof函数返回操作数在内存中的字节大小，参数可以是任意类型的表达式，但是它并不会对表达式进行求值。一个Sizeof函数调用是一个对应uintptr类型的常量表达式，因此返回的结果可以用作数组类型的长度大小，或者用作计算其他的常量。

import (  
 "fmt"  
 "unsafe"  
)  
  
func main() {  
 fmt.Println(unsafe.Sizeof(float64(0))) // "8"  
}

Sizeof函数返回的大小只包括数据结构中固定的部分，例如字符串对应结构体中的指针和字符串长度部分，但是并不包含指针指向的字符串的内容。Go语言中非聚合类型通常有一个固定的大小，尽管在不同工具链下生成的实际大小可能会有所不同。考虑到可移植性，引用类型或包含引用类型的大小在32位平台上是4个字节，在64位平台上是8个字节。

计算机在加载和保存数据时，如果内存地址合理地对齐的将会更有效率。例如2字节大小的int16类型的变量地址应该是偶数，一个4字节大小的rune类型变量的地址应该是4的倍数，一个8字节大小的float64、uint64或64-bit指针类型变量的地址应该是8字节对齐的。但是对于再大的地址对齐倍数则是不需要的，即使是complex128等较大的数据类型最多也只是8字节对齐。

由于地址对齐这个因素，一个聚合类型（结构体或数组）的大小至少是所有字段或元素大小的总和，或者更大因为可能存在内存空洞。**内存空洞**是编译器自动添加的没有被使用的内存空间，用于保证后面每个字段或元素的地址相对于结构或数组的开始地址能够合理地对齐（译注：内存空洞可能会存在一些随机数据，可能会对用unsafe包直接操作内存的处理产生影响）。

| **类型** | **大小** |
| --- | --- |
| bool | 1个字节 |
| intN, uintN, floatN, complexN | N/8个字节（例如float64是8个字节） |
| int, uint, uintptr | 1个机器字 |
| \*T | 1个机器字 |
| string | 2个机器字（data、len） |
| []T | 3个机器字（data、len、cap） |
| map | 1个机器字 |
| func | 1个机器字 |
| chan | 1个机器字 |
| interface | 2个机器字（type、value） |

Go语言的规范并没有要求一个字段的声明顺序和内存中的顺序是一致的，所以理论上一个编译器可以随意地重新排列每个字段的内存位置，虽然在写作本书的时候编译器还没有这么做。下面的三个结构体虽然有着相同的字段，但是第一种写法比另外的两个需要多50%的内存。

关于内存地址对齐算法的细节超出了本书的范围，也不是每一个结构体都需要担心这个问题，不过有效的包装可以使数据结构更加紧凑（译注：未来的Go语言编译器应该会默认优化结构体的顺序，当然应该也能够指定具体的内存布局，相同讨论请参考 [Issue10014](https://github.com/golang/go/issues/10014" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) ），内存使用率和性能都可能会受益。

unsafe.Alignof 函数返回对应参数的类型需要对齐的倍数。和 Sizeof 类似， Alignof 也是返回一个常量表达式，对应一个常量。通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。

unsafe.Offsetof 函数的参数必须是一个字段 x.f，然后返回 f 字段相对于 x 起始地址的偏移量，包括可能的空洞。

图 13.1 显示了一个结构体变量 x 以及其在32位和64位机器上的典型的内存。灰色区域是空洞。

var x struct {

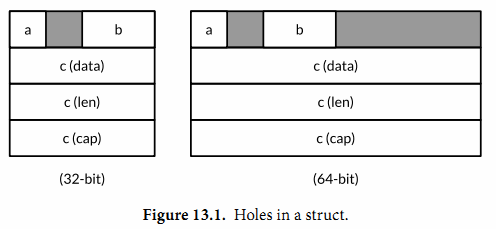
a bool

b int16

c []int

}

下面显示了对x和它的三个字段调用unsafe包相关函数的计算结果：



32位系统：

Sizeof(x) = 16 Alignof(x) = 4

Sizeof(x.a) = 1 Alignof(x.a) = 1 Offsetof(x.a) = 0

Sizeof(x.b) = 2 Alignof(x.b) = 2 Offsetof(x.b) = 2

Sizeof(x.c) = 12 Alignof(x.c) = 4 Offsetof(x.c) = 4

64位系统：

Sizeof(x) = 32 Alignof(x) = 8

Sizeof(x.a) = 1 Alignof(x.a) = 1 Offsetof(x.a) = 0

Sizeof(x.b) = 2 Alignof(x.b) = 2 Offsetof(x.b) = 2

Sizeof(x.c) = 24 Alignof(x.c) = 8 Offsetof(x.c) = 8

虽然这几个函数在不安全的unsafe包，但是这几个函数调用并不是真的不安全，特别在需要优化内存空间时它们返回的结果对于理解原生的内存布局很有帮助。

## **2. unsafe.Pointer**

大多数指针类型会写成\*T，表示是“一个指向T类型变量的指针”。unsafe.Pointer是特别定义的一种指针类型（译注：类似C语言中的void\*类型的指针），它可以包含任意类型变量的地址。当然，我们不可以直接通过\*p来获取unsafe.Pointer指针指向的真实变量的值，因为我们并不知道变量的具体类型。和普通指针一样，unsafe.Pointer指针也是可以比较的，并且支持和nil常量比较判断是否为空指针。

一个普通的\*T类型指针可以被转化为unsafe.Pointer类型指针，并且一个unsafe.Pointer类型指针也可以被转回普通的指针，被转回普通的指针类型并不需要和原始的\*T类型相同。通过将\*float64类型指针转化为\*uint64类型指针，我们可以查看一个浮点数变量的位模式。

package math  
  
import (  
 "fmt"  
 "unsafe"  
)  
  
func Float64bits(f float64) uint64 {   
 return \*(\*uint64)(unsafe.Pointer(&f))   
}  
  
func main() {  
 fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"  
}

通过转为新类型指针，我们可以更新浮点数的位模式。通过位模式操作浮点数是可以的，但是更重要的意义是指针转换语法让我们可以在不破坏类型系统的前提下向内存写入任意的值。

一个unsafe.Pointer指针也可以被转化为uintptr类型，然后保存到指针型数值变量中（译注：这只是和当前指针相同的一个数字值，并不是一个指针），然后用以做必要的指针数值运算。（第三章内容，uintptr是一个无符号的整型数，足以保存一个地址）这种转换虽然也是可逆的，但是将uintptr转为unsafe.Pointer指针可能会破坏类型系统，因为并不是所有的数字都是有效的内存地址。

许多将unsafe.Pointer指针转为原生数字，然后再转回为unsafe.Pointer类型指针的操作也是不安全的。比如下面的例子需要将变量x的地址加上b字段地址偏移量转化为\*int16类型指针，然后通过该指针更新x.b：

*gopl.io/ch13/unsafeptr*

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"

上面的写法尽管很繁琐，但在这里并不是一件坏事，因为这些功能应该很谨慎地使用。不要试图引入一个uintptr类型的临时变量，因为它可能会破坏代码的安全性（译注：这是真正可以体会unsafe包为何不安全的例子）。下面段代码是错误的：

// NOTE: subtly incorrect!  
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)  
pb := (\*int16)(unsafe.Pointer(tmp))  
\*pb = 42

产生错误的原因很微妙。有时候垃圾回收器会移动一些变量以降低内存碎片等问题。这类垃圾回收器被称为**移动GC**。当一个变量被移动，所有的保存该变量旧地址的指针必须同时被更新为变量移动后的新地址。从垃圾收集器的视角来看，一个unsafe.Pointer是一个指向变量的指针，因此当变量被移动时对应的指针也必须被更新；但是uintptr类型的临时变量只是一个普通的数字，所以其值不应该被改变。上面错误的代码因为引入一个非指针的临时变量tmp，导致垃圾收集器无法正确识别这个是一个指向变量x的指针。当第二个语句执行时，变量x可能已经被转移，这时候临时变量tmp也就不再是现在的&x.b地址。第三个向之前无效地址空间的赋值语句将彻底摧毁整个程序！

还有很多类似原因导致的错误。例如这条语句：

pT := uintptr(unsafe.Pointer(new(T))) // 提示: 错误!

这里并没有指针引用new新创建的变量，因此该语句执行完成之后，垃圾收集器有权马上回收其内存空间，所以返回的pT将是无效的地址。

虽然目前的Go语言实现还没有使用移动GC（译注：未来可能实现），但这不该是编写错误代码侥幸的理由：当前的Go语言实现已经有移动变量的场景。在5.2节我们提到goroutine的栈是根据需要动态增长的。当发生栈动态增长的时候，原来栈中的所有变量可能需要被移动到新的更大的栈中，所以我们并不能确保变量的地址在整个使用周期内是不变的。

在编写本文时，还没有清晰的原则来指引Go程序员，什么样的unsafe.Pointer和uintptr的转换是不安全的（参考 [Issue7192](https://github.com/golang/go/issues/7192" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) ）. 译注: 该问题已经关闭），因此我们强烈建议按照最坏的方式处理。将所有包含变量地址的uintptr类型变量当作BUG处理，同时减少不必要的unsafe.Pointer类型到uintptr类型的转换。在第一个例子中，有三个转换——字段偏移量到uintptr的转换和转回unsafe.Pointer类型的操作——所有的转换全在一个表达式完成。

当调用一个库函数，并且返回的是uintptr类型地址时（译注：普通方法实现的函数尽量不要返回该类型。下面例子是reflect包的函数，reflect包和unsafe包一样都是采用特殊技术实现的，编译器可能给它们开了后门），比如下面反射包中的相关函数，返回的结果应该立即转换为unsafe.Pointer以确保指针指向的是相同的变量。

package reflect

func (Value) Pointer() uintptr

func (Value) UnsafeAddr() uintptr

func (Value) InterfaceData() [2]uintptr // (index 1)

## **3. 示例: 深度相等判断**

来自reflect包的DeepEqual函数可以对两个值进行深度相等判断。DeepEqual函数使用内建的==比较操作符对基础类型进行相等判断，对于复合类型则递归该变量的每个基础类型然后做类似的比较判断。因为它可以工作在任意的类型上，甚至对于一些不支持==操作运算符的类型也可以工作，因此在一些测试代码中广泛地使用该函数。比如下面的代码是用DeepEqual函数比较两个字符串slice是否相等。

func TestSplit(t \*testing.T) {  
 got := strings.Split("a:b:c", ":")  
 want := []string{"a", "b", "c"};  
 if !reflect.DeepEqual(got, want) { /\* ... \*/ }  
}

尽管DeepEqual函数很方便，而且可以支持任意的数据类型，但是它也有不足之处。例如，它将一个nil值的map和非nil值但是空的map视作不相等，同样nil值的slice 和非nil但是空的slice也视作不相等。

var a, b []string = nil, []string{}  
fmt.Println(reflect.DeepEqual(a, b)) // "false"  
var c, d map[string]int = nil, make(map[string]int)  
fmt.Println(reflect.DeepEqual(c, d)) // "false"

我们希望在这里实现一个自己的Equal函数，用于比较类型的值。和DeepEqual函数类似的地方是它也是基于slice和map的每个元素进行递归比较，不同之处是它将nil值的slice（map类似）和非nil值但是空的slice视作相等的值。基础部分的比较可以基于reflect包完成，和12.3章的Display函数的实现方法类似。同样，我们也定义了一个内部函数equal，用于内部的递归比较。读者目前不用关心seen参数的具体含义。对于每一对需要比较的x和y，equal函数首先检测它们是否都有效（或都无效），然后检测它们是否是相同的类型。剩下的部分是一个巨大的switch分支，用于相同基础类型的元素比较。因为页面空间的限制，我们省略了一些相似的分支。

*gopl.io/ch13/equal*

func equal(x, y reflect.Value, seen map[comparison]bool) bool {  
 if !x.IsValid() || !y.IsValid() {  
 return x.IsValid() == y.IsValid()  
 }  
 if x.Type() != y.Type() {  
 return false  
 }  
  
 // ...cycle check omitted (shown later)...  
  
 switch x.Kind() {  
 case reflect.*Bool*:  
 return x.Bool() == y.Bool()  
 case reflect.*String*:  
 return x.String() == y.String()  
  
 // ...numeric cases omitted for brevity...  
  
 case reflect.*Chan*, reflect.*UnsafePointer*, reflect.*Func*:  
 return x.Pointer() == y.Pointer()  
 case reflect.*Ptr*, reflect.*Interface*:  
 return equal(x.Elem(), y.Elem(), seen)  
 case reflect.*Array*, reflect.*Slice*:  
 if x.Len() != y.Len() {  
 return false  
 }  
 for i := 0; i < x.Len(); i++ {  
 if !equal(x.Index(i), y.Index(i), seen) {  
 return false  
 }  
 }  
 return true  
 // ...struct and map cases omitted for brevity...  
 }  
 panic("unreachable")  
}

和前面的建议一样，我们并不公开reflect包相关的接口，所以导出的函数需要在内部自己将变量转为reflect.Value类型。

// Equal reports whether x and y are deeply equal.  
func Equal(x, y interface{}) bool {  
 seen := make(map[comparison]bool)  
 return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)  
}  
type comparison struct {  
 x, y unsafe.Pointer  
 treflect.Type  
}

为了确保算法对于有环的数据结构也能正常退出，我们必须记录每次已经比较的变量，从而避免进入第二次的比较。Equal函数分配了一组用于比较的结构体，包含每对比较对象的地址（unsafe.Pointer形式保存）和类型。我们要记录类型的原因是，有些不同的变量可能对应相同的地址。例如，如果x和y都是数组类型，那么x和x[0]将对应相同的地址，y和y[0]也是对应相同的地址，这可以用于区分x与y之间的比较或x[0]与y[0]之间的比较是否进行过了。

// cycle check  
if x.CanAddr() && y.CanAddr() {  
 xptr := unsafe.Pointer(x.UnsafeAddr())  
 yptr := unsafe.Pointer(y.UnsafeAddr())  
 if xptr == yptr {  
 return true // identical references  
 }  
 c := comparison{xptr, yptr, x.Type()}  
 if seen[c] {  
 return true // already seen  
 }  
 seen[c] = true  
}

这是Equal函数用法的例子:

fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3})) // "true"  
fmt.Println(Equal([]string{"foo"}, []string{"bar"})) // "false"  
fmt.Println(Equal([]string(nil), []string{})) // "true"  
fmt.Println(Equal(map[string]int(nil), map[string]int{})) // "true"

Equal函数甚至可以处理类似12.3章中导致Display陷入死循环的带有环的数据。

// Circular linked lists a -> b -> a and c -> c.  
type link struct {  
 value string  
 tail \*link  
}  
  
func main() {  
 a, b, c := &link{value: "a"}, &link{value: "b"}, &link{value: "c"}  
 a.tail, b.tail, c.tail = b, a, c  
 fmt.Println(Equal(a, a)) // "true"  
 fmt.Println(Equal(b, b)) // "true"  
 fmt.Println(Equal(c, c)) // "true"  
 fmt.Println(Equal(a, b)) // "false"  
 fmt.Println(Equal(a, c)) // "false"  
}

****练习 13.1：**** 定义一个深比较函数，对于十亿以内的数字比较，忽略类型差异。

****练习 13.2：**** 编写一个函数，报告其参数是否为循环数据结构。

## **4. 通过cgo调用C代码**

Go程序可能会遇到要访问C语言的某些硬件驱动函数的场景，或者是从一个C++语言实现的嵌入式数据库查询记录的场景，或者是使用Fortran语言实现的一些线性代数库的场景。C语言作为一个通用语言，很多库会选择提供一个C兼容的API，然后用其他不同的编程语言实现（译者：Go语言需要也应该拥抱这些巨大的代码遗产）。

在本节中，我们将构建一个简易的数据压缩程序，使用了一个Go语言自带的叫cgo的用于支援C语言函数调用的工具。这类工具一般被称为 *foreign-function interfaces* （简称ffi），并且在类似工具中cgo也不是唯一的。SWIG（[http://swig.org](http://swig.org/" \t "http://books.studygolang.com/gopl-zh/ch13/_blank)）是另一个类似的且被广泛使用的工具，SWIG提供了很多复杂特性以支援C++的特性，但SWIG并不是我们要讨论的主题。

在标准库的compress/...子包有很多流行的压缩算法的编码和解码实现，包括流行的LZW压缩算法（Unix的compress命令用的算法）和DEFLATE压缩算法（GNU gzip命令用的算法）。这些包的API的细节虽然有些差异，但是它们都提供了针对 io.Writer类型输出的压缩接口和提供了针对io.Reader类型输入的解压缩接口。例如：

package gzip // compress/gzip

func NewWriter(w io.Writer) io.WriteCloser

func NewReader(r io.Reader) (io.ReadCloser, error)

bzip2压缩算法，是基于优雅的Burrows-Wheeler变换算法，运行速度比gzip要慢，但是可以提供更高的压缩比。标准库的compress/bzip2包目前还没有提供bzip2压缩算法的实现。完全从头开始实现一个压缩算法是一件繁琐的工作，而且 [http://bzip.org](http://bzip.org/" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) 已经有现成的libbzip2的开源实现，不仅文档齐全而且性能又好。

如果是比较小的C语言库，我们完全可以用纯Go语言重新实现一遍。如果我们对性能也没有特殊要求的话，我们还可以用os/exec包的方法将C编写的应用程序作为一个子进程运行。只有当你需要使用复杂而且性能更高的底层C接口时，就是使用cgo的场景了（译注：用os/exec包调用子进程的方法会导致程序运行时依赖那个应用程序）。下面我们将通过一个例子讲述cgo的具体用法。

译注：本章采用的代码都是最新的。因为之前已经出版的书中包含的代码只能在Go1.5之前使用。从Go1.6开始，Go语言已经明确规定了哪些Go语言指针可以直接传入C语言函数。新代码重点是增加了bz2alloc和bz2free的两个函数，用于bz\_stream对象空间的申请和释放操作。下面是新代码中增加的注释，说明这个问题：

// The version of this program that appeared in the first and second

// printings did not comply with the proposed rules for passing

// pointers between Go and C, described here:

// https://github.com/golang/proposal/blob/master/design/12416-cgo-pointers.md

//

// The rules forbid a C function like bz2compress from storing 'in'

// and 'out' (pointers to variables allocated by Go) into the Go

// variable 's', even temporarily.

//

// The version below, which appears in the third printing, has been

// corrected. To comply with the rules, the bz\_stream variable must

// be allocated by C code. We have introduced two C functions,

// bz2alloc and bz2free, to allocate and free instances of the

// bz\_stream type. Also, we have changed bz2compress so that before

// it returns, it clears the fields of the bz\_stream that contain

// pointers to Go variables.

要使用libbzip2，我们需要先构建一个bz\_stream结构体，用于保持输入和输出缓存。然后有三个函数：BZ2\_bzCompressInit用于初始化缓存，BZ2\_bzCompress用于将输入缓存的数据压缩到输出缓存，BZ2\_bzCompressEnd用于释放不需要的缓存。（目前不要担心包的具体结构，这个例子的目的就是演示各个部分如何组合在一起的。）

我们可以在Go代码中直接调用BZ2\_bzCompressInit和BZ2\_bzCompressEnd，但是对于BZ2\_bzCompress，我们将定义一个C语言的包装函数，用它完成真正的工作。下面是C代码，对应一个独立的文件。

*gopl.io/ch13/bzip*

/\* This file is gopl.io/ch13/bzip/bzip2.c, \*//\* a simple wrapper for libbzip2 suitable for cgo. \*/#include <bzlib.h>

int bz2compress(bz\_stream \*s, int action,

char \*in, unsigned \*inlen, char \*out, unsigned \*outlen) {

s->next\_in = in;

s->avail\_in = \*inlen;

s->next\_out = out;

s->avail\_out = \*outlen;

int r = BZ2\_bzCompress(s, action);

\*inlen -= s->avail\_in;

\*outlen -= s->avail\_out;

s->next\_in = s->next\_out = NULL;

return r;

}

现在让我们转到Go语言部分，第一部分如下所示。其中import "C"的语句是比较特别的。其实并没有一个叫C的包，但是这行语句会让Go编译程序在编译之前先运行cgo工具。

// Package bzip provides a writer that uses bzip2 compression (bzip.org).package bzip

/\*

#cgo CFLAGS: -I/usr/include

#cgo LDFLAGS: -L/usr/lib -lbz2

#include <bzlib.h>

#include <stdlib.h>

bz\_stream\* bz2alloc() { return calloc(1, sizeof(bz\_stream)); }

int bz2compress(bz\_stream \*s, int action,

char \*in, unsigned \*inlen, char \*out, unsigned \*outlen);

void bz2free(bz\_stream\* s) { free(s); }

\*/import "C"

import (

"io"

"unsafe"

)

type writer struct {

w io.Writer // underlying output stream

stream \*C.bz\_stream

outbuf [64 \* 1024]byte

}

// NewWriter returns a writer for bzip2-compressed streams.func NewWriter(out io.Writer) io.WriteCloser {

const blockSize = 9

const verbosity = 0

const workFactor = 30

w := &writer{w: out, stream: C.bz2alloc()}

C.BZ2\_bzCompressInit(w.stream, blockSize, verbosity, workFactor)

return w

}

在预处理过程中，cgo工具生成一个临时包用于包含所有在Go语言中访问的C语言的函数或类型。例如C.bz\_stream和C.BZ2\_bzCompressInit。cgo工具通过以某种特殊的方式调用本地的C编译器来发现在Go源文件导入声明前的注释中包含的C头文件中的内容（译注：import "C"语句前紧挨着的注释是对应cgo的特殊语法，对应必要的构建参数选项和C语言代码）。

在cgo注释中还可以包含#cgo指令，用于给C语言工具链指定特殊的参数。例如CFLAGS和LDFLAGS分别对应传给C语言编译器的编译参数和链接器参数，使它们可以从特定目录找到bzlib.h头文件和libbz2.a库文件。这个例子假设你已经在/usr目录成功安装了bzip2库。如果bzip2库是安装在不同的位置，你需要更新这些参数（译注：这里有一个从纯C代码生成的cgo绑定，不依赖bzip2静态库和操作系统的具体环境，具体请访问 [https://github.com/chai2010/bzip2](https://github.com/chai2010/bzip2" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) ）。

NewWriter函数通过调用C语言的BZ2\_bzCompressInit函数来初始化stream中的缓存。在writer结构中还包括了另一个buffer，用于输出缓存。

下面是Write方法的实现，返回成功压缩数据的大小，主体是一个循环中调用C语言的bz2compress函数实现的。从代码可以看到，Go程序可以访问C语言的bz\_stream、char和uint类型，还可以访问bz2compress等函数，甚至可以访问C语言中像BZ\_RUN那样的宏定义，全部都是以C.x语法访问。其中C.uint类型和Go语言的uint类型并不相同，即使它们具有相同的大小也是不同的类型。

func (w \*writer) Write(data []byte) (int, error) {

if w.stream == nil {

panic("closed")

}

var total int // uncompressed bytes written

for len(data) > 0 {

inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))

C.bz2compress(w.stream, C.BZ\_RUN,

(\*C.char)(unsafe.Pointer(&data[0])), &inlen,

(\*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)

total += int(inlen)

data = data[inlen:]

if \_, err := w.w.Write(w.outbuf[:outlen]); err != nil {

return total, err

}

}

return total, nil

}

在循环的每次迭代中，向bz2compress传入数据的地址和剩余部分的长度，还有输出缓存w.outbuf的地址和容量。这两个长度信息通过它们的地址传入而不是值传入，因为bz2compress函数可能会根据已经压缩的数据和压缩后数据的大小来更新这两个值。每个块压缩后的数据被写入到底层的io.Writer。

Close方法和Write方法有着类似的结构，通过一个循环将剩余的压缩数据刷新到输出缓存。

// Close flushes the compressed data and closes the stream.// It does not close the underlying io.Writer.func (w \*writer) Close() error {

if w.stream == nil {

panic("closed")

}

defer func() {

C.BZ2\_bzCompressEnd(w.stream)

C.bz2free(w.stream)

w.stream = nil

}()

for {

inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))

r := C.bz2compress(w.stream, C.BZ\_FINISH, nil, &inlen,

(\*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)

if \_, err := w.w.Write(w.outbuf[:outlen]); err != nil {

return err

}

if r == C.BZ\_STREAM\_END {

return nil

}

}

}

压缩完成后，Close方法用了defer函数确保函数退出前调用C.BZ2\_bzCompressEnd和C.bz2free释放相关的C语言运行时资源。此刻w.stream指针将不再有效，我们将它设置为nil以保证安全，然后在每个方法中增加了nil检测，以防止用户在关闭后依然错误使用相关方法。

上面的实现中，不仅仅写是非并发安全的，甚至并发调用Close和Write方法也可能导致程序的的崩溃。修复这个问题是练习13.3的内容。

下面的bzipper程序，使用我们自己包实现的bzip2压缩命令。它的行为和许多Unix系统的bzip2命令类似。

*gopl.io/ch13/bzipper*

// Bzipper reads input, bzip2-compresses it, and writes it out.package main

import (

"io"

"log"

"os"

"gopl.io/ch13/bzip"

)

func main() {

w := bzip.NewWriter(os.Stdout)

if \_, err := io.Copy(w, os.Stdin); err != nil {

log.Fatalf("bzipper: %v\n", err)

}

if err := w.Close(); err != nil {

log.Fatalf("bzipper: close: %v\n", err)

}

}

在上面的场景中，我们使用bzipper压缩了/usr/share/dict/words系统自带的词典，从938,848字节压缩到335,405字节。大约是原始数据大小的三分之一。然后使用系统自带的bunzip2命令进行解压。压缩前后文件的SHA256哈希码是相同了，这也说明了我们的压缩工具是正确的。（如果你的系统没有sha256sum命令，那么请先按照练习4.2实现一个类似的工具）

$ go build gopl.io/ch13/bzipper

$ wc -c < /usr/share/dict/words

938848

$ sha256sum < /usr/share/dict/words

126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -

$ ./bzipper < /usr/share/dict/words | wc -c

335405

$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum

126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -

我们演示了如何将一个C语言库链接到Go语言程序。相反，将Go编译为静态库然后链接到C程序，或者将Go程序编译为动态库然后在C程序中动态加载也都是可行的（译注：在Go1.5中，Windows系统的Go语言实现并不支持生成C语言动态库或静态库的特性。不过好消息是，目前已经有人在尝试解决这个问题，具体请访问 [Issue11058](https://github.com/golang/go/issues/11058" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) ）。这里我们只展示的cgo很小的一些方面，更多的关于内存管理、指针、回调函数、中断信号处理、字符串、errno处理、终结器，以及goroutines和系统线程的关系等，有很多细节可以讨论。特别是如何将Go语言的指针传入C函数的规则也是异常复杂的（译注：简单来说，要传入C函数的Go指针指向的数据本身不能包含指针或其他引用类型；并且C函数在返回后不能继续持有Go指针；并且在C函数返回之前，Go指针是被锁定的，不能导致对应指针数据被移动或栈的调整），部分的原因在13.2节有讨论到，但是在Go1.5中还没有被明确（译注：Go1.6将会明确cgo中的指针使用规则）。如果要进一步阅读，可以从 [https://golang.org/cmd/cgo](https://golang.org/cmd/cgo" \t "http://books.studygolang.com/gopl-zh/ch13/_blank) 开始。

****练习 13.3：**** 使用sync.Mutex以保证bzip2.writer在多个goroutines中被并发调用是安全的。

****练习 13.4：**** 因为C库依赖的限制。 使用os/exec包启动/bin/bzip2命令作为一个子进程，提供一个纯Go的bzip.NewWriter的替代实现（译注：虽然是纯Go实现，但是运行时将依赖/bin/bzip2命令，其他操作系统可能无法运行）。

## **5. 几点忠告**

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。

高级语言使得程序员不用再关心真正运行程序的指令细节，同时也不再需要关注许多如内存布局之类的实现细节。因为高级语言这个绝缘的抽象层，我们可以编写安全健壮的，并且可以运行在不同操作系统上的具有高度可移植性的程序。

但是unsafe包，它让程序员可以透过这个绝缘的抽象层直接使用一些必要的功能，虽然可能是为了获得更好的性能。但是代价就是牺牲了可移植性和程序安全，因此使用unsafe包是一个危险的行为。我们对何时以及如何使用unsafe包的建议和我们在11.5节提到的Knuth对过早优化的建议类似。大多数Go程序员可能永远不会需要直接使用unsafe包。当然，也永远都会有一些需要使用unsafe包实现会更简单的场景。如果确实认为使用unsafe包是最理想的方式，那么应该尽可能将它限制在较小的范围，这样其它代码就可以忽略unsafe的影响。

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。

最后，用Go快乐地编程。我们希望你能像我们一样喜欢Go语言。