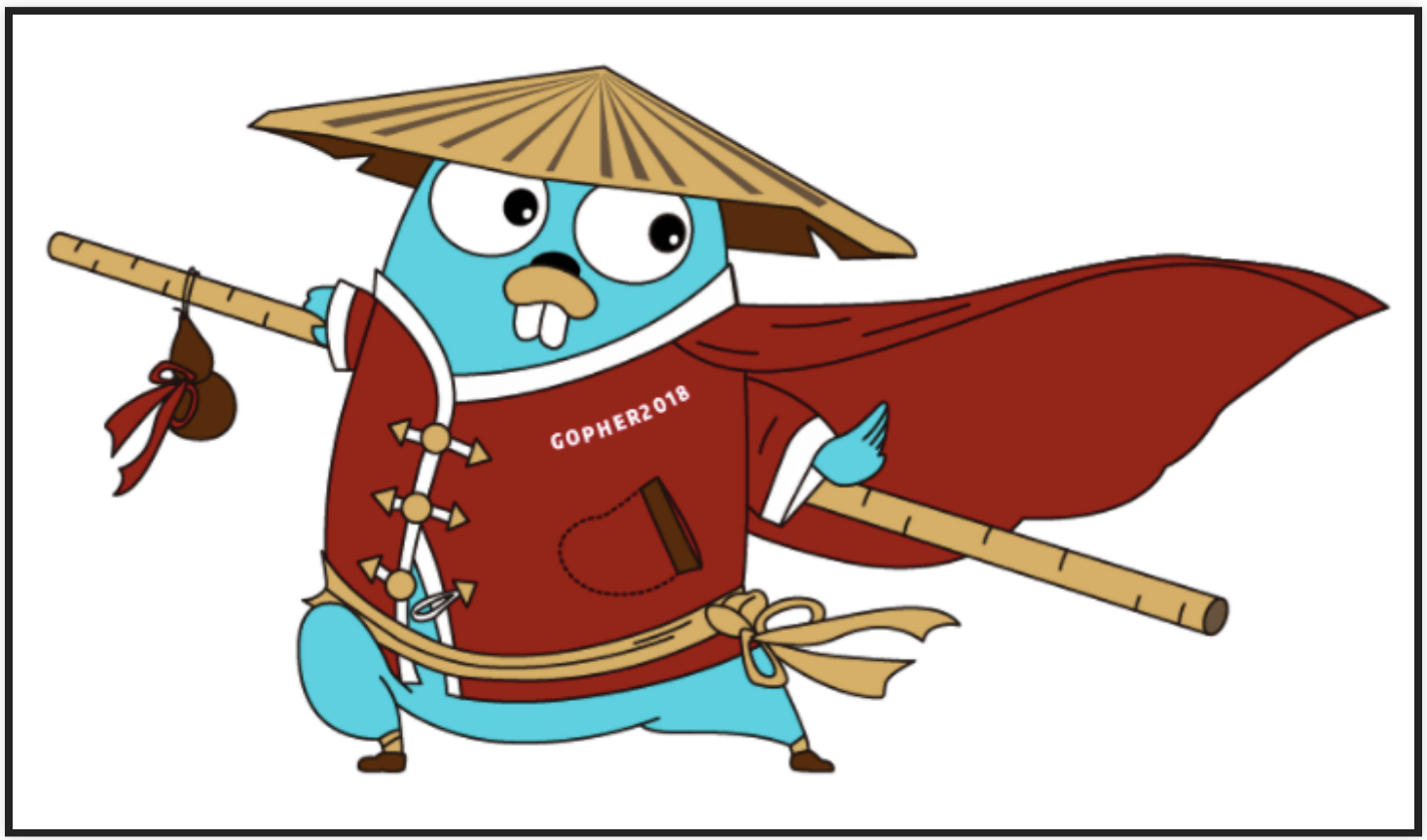
GopherChina2018

深入CGO编程

chai2010 (柴树杉)

@青云QingCloud

感谢 GopherChina



http://gopherchina.org/

感谢 韦光京 对CGO的贡献

ld: add one empty symbol into pe to make dumpbin works.	1ac7a69	Wei Guangjing <vcc.163@gmail.com>	Aug 10, 2011 at 9:09 AM
build: fixes for mingw-w64	ee14fbd	Wei Guangjing <vcc.163@gmail.com>	Jul 26, 2011 at 1:39 AM
ld: don't skip first 2 symbols in ldpe. some object files don't has file name symbol.	0871af2	Wei Guangjing <vcc.163@gmail.com>	Jul 26, 2011 at 12:25 AM
ld: fixes .bss for ldpe	7ce1a4b	Wei Guangjing <vcc.163@gmail.com>	Jul 24, 2011 at 12:21 AM
ld: fixes ldpe link with SXREF global values.	4e5e12e	Wei Guangjing <vcc.163@gmail.com>	Jul 23, 2011 at 9:21 AM
cgo: windows amd64 port	9f63659	Wei Guangjing <vcc.163@gmail.com>	Jul 19, 2011 at 10:47 PM
net: windows/amd64 port	3505045	Wei Guangjing <vcc.163@gmail.com>	Jul 14, 2011 at 3:44 AM
runtime: stdcall_raw stack 16byte align for Win64	a6e6091	Wei Guangjing <vcc.163@gmail.com>	Jul 14, 2011 at 2:44 AM
debug/pe: fixes ImportedSymbols for Win64.	f340b3d	Wei Guangjing <vcc.163@gmail.com>	Jul 13, 2011 at 2:29 AM
windows: define and use syscall.Handle Fixes #1487.	63b8b94	Wei Guangjing <vcc.163@gmail.com>	Jul 1, 2011 at 10:18 PM
runtime: windows/amd64 port	f83609f	Wei Guangjing <vcc.163@gmail.com>	Jun 29, 2011 at 3:37 PM
ld: fix link Windows PE __declspec(dllimport) symbol	9e2ffc3	Wei Guangjing <vcc.163@gmail.com>	Jun 14, 2011 at 11:05 PM
8l: emit resources (.rsrc) in Windows PE.	2ad58d4	Wei Guangjing <vcc.163@gmail.com>	May 25, 2011 at 7:53 PM
6l: fix emit windows dwarf sections	b256358	Wei Guangjing <vcc.163@gmail.com>	May 19, 2011 at 1:12 AM
6l: pe fixes	8ba4df2	Wei Guangjing <vcc.163@gmail.com>	Feb 18, 2011 at 7:58 AM
debug/pe: ImportedSymbols fixes	44bcc1f	Wei Guangjing <vcc.163@gmail.com>	Feb 10, 2011 at 11:22 PM
fix windows build	6606995	Wei Guangjing <vcc.163@gmail.com>	Jan 29, 2011 at 12:44 PM
8l: add PE dynexport	a3120f6	Wei Guangjing <vcc.163@gmail.com>	Jan 27, 2011 at 9:26 PM
8l: fix ldpe sym name length == 8 strdup incorrect.	3542199	Wei Guangjing <vcc.163@gmail.com>	Jan 27, 2011 at 9:26 PM
8l: emit DWARF in Windows PE.	f1751e7	Wei Guangjing <vcc.163@gmail.com>	Jan 21, 2011 at 12:28 AM
cgo: windows/386 port	1aa2d88	Wei Guangjing <vcc.163@gmail.com>	Jan 20, 2011 at 11:22 PM
6l: windows/amd64 port	3aec551	Wei Guangjing <vcc.163@gmail.com>	Jan 20, 2011 at 10:21 PM
net: implement windows timeout	ff25900	Wei Guangjing <vcc.163@gmail.com>	Jan 20, 2011 at 3:49 AM
Fix windows build.	e04ef77	Wei Guangjing <vcc.163@gmail.com>	Dec 13, 2010 at 1:41 PM
8l : add dynimport to import table in Windows PE, initial make cgo dll work.	70deac6	Wei Guangjing <vcc.163@gmail.com>	Dec 8, 2010 at 4:28 AM
net: add ReadFrom and WriteTo windows version.	95c341f	Wei Guangjing <vcc.163@gmail.com>	Nov 23, 2010 at 12:01 AM
net: fix windows build	11ace8e	Wei Guangjing <vcc.163@gmail.com>	Nov 6, 2010 at 11:08 AM
debug/pe, cgo: add windows support	035696c	Wei Guangjing <vcc.163@gmail.com>	Nov 2, 2010 at 5:52 AM
8l: fix windows build.	77eb94c	Wei Guangjing <vcc.163@gmail.com>	Oct 16, 2010 at 11:37 AM
syscall: implement WaitStatus and Wait4() for windows	d3a2118	Wei Guangjing <vcc.163@gmail.com>	Oct 12, 2010 at 12:42 PM
libcgo: set g, m in thread local storage for windows 386.	6a624fa	Wei Guangjing <vcc.163@gmail.com>	Sep 27, 2010 at 9:44 PM
net: implement windows version of LookupHost/Port/SRV	adc13b1	Wei Guangjing <vcc.163@gmail.com>	Jul 29, 2010 at 12:58 PM
syscall: add windows version of Pipe()	ad4f95d	Wei Guangjing <vcc.163@gmail.com>	Jul 26, 2010 at 1:55 PM

https://github.com/golang/go/commits?author=wgj-zz

幻灯片 网址

https://chai2010.cn/talks/cgo2018/

个人简介

- @青云QingCloud, 应用平台研发工程师
- Go语言代码 贡献者(ChaiShushan)
- Go语言圣经 翻译者
- Go语言高级编程 作者(开发中...)
- OpenPitrix 多云应用管理平台开发者
- https://github.com/chai2010
- https://chai2010.cn

珠三角技术沙龙深圳(2011.02.27)

Go集成C&C++代码

珠三角技术沙龙深圳(2011.02.27)

更多图片

个人签名

- 当歌曲、传说都已经缄默的时候，只有代码还在说话!
- Less is more!

内容大纲

- CGO的价值

- 快速入门
- 类型转换
- 函数调用
- CGO内部机制
- 实战: 包装 `C.qsort`
- 内存模型

内容大纲(续)

- Go访问C++对象, Go对象导出为C++对象
- 静态库和动态库
- 编写Python扩展
- 编译和链接参数

CGO的价值

- 小调查: 有多少人 **听说过** 或 **简单使用过** CGO?

---

1. 没有银弹, Go语言也不是银弹, 无法解决全部问题
2. 通过CGO可以继承C/C++将近半个世纪的软件积累
3. 通过CGO可以用Go给其它系统写C接口的共享库
4. CGO是Go和其它语言直接通讯的桥梁

- CGO 是一个保底的后备技术
- CGO 是 Go 的替补技术

可能的CGO的场景

- 通过OpenGL或OpenCL使用显卡的计算能力
- 通过OpenCV来进行图像分析
- 通过Go编写Python扩展
- 通过Go编写移动应用

Cgo is not Go

---

https://dave.cheney.net/2016/01/18/cgo-is-not-go

快速入门

examples/hello-v1/main.go:

```
package main

//#include <stdio.h>
import "C"

func main() {
    C.puts(C.CString("你好, GopherChina 2018!\n"))
}
```

编译运行:

```
$ go run examples/hello-v1/main.go
你好, GopherChina 2018!
```

简单说明

- `import "C"` 表示启用 CGO
- `import "C"` 前的注释表示包含C头文件: `<stdio.h>`
- `C.CString` 表示将 Go 字符串转为 C 字符串
- `C.puts` 调用C语言的puts函数输出 C 字符串

调用自定义的C函数

examples/hello-v2/main.go:

```
package main

/*
#include <stdio.h>

static void SayHello(const char* s) {
    puts(s);
}
*/
import "C"

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}
```

C代码模块化

examples/hello-v3/hello.h:

```
extern void SayHello(const char* s);
```

examples/hello-v3/hello.c:

```
#include "hello.h"

#include <stdio.h>

void SayHello(const char* s) {
    puts(s);
}
```

examples/hello-v3/main.go:

```
//#include "./hello.h"
import "C"
```

C代码模块化 - 改用Go重写C模块

examples/hello-v4/hello.h:

```
extern void SayHello(/* const */ char* s);
```

examples/hello-v4/hello.go:

```
package main

import "C"
import "fmt"

//export SayHello
func SayHello(s *C.char) {
    fmt.Print(C.GoString(s))
}
```

- 函数参数去掉 const 修饰符
- hello.c => hello.go

手中无剑, 心中有剑

examples/hello-v5/hello.go:

```
package main

// extern void SayHello(char* s);
import "C"
import "fmt"

func main() {
    C.SayHello(C.CString("Hello, World\n"))
}

//export SayHello
func SayHello(s *C.char) {
    fmt.Print(C.GoString(s))
}
```

- C 语言版本 SayHello 函数实现只存在于心中
- 面向纯 C 接口的 Go 语言编程

忘掉心中之剑

```
// +build go1.10

package main

// extern void SayHello(_GoString_ s);
import "C"
import "fmt"

func main() {
    C.SayHello("Hello, World\n")
}

//export SayHello
func SayHello(s string) {
    fmt.Print(s)
```

- GoString 也是一种 C 字符串
- Go的一切都可以从C理解

类型转换

- 指针 - unsafe 包的灵魂
- Go字符串和切片的结构

- Go指针和C指针之间的转换
- 数值和指针之间的转换
- 不同类型指针转换
- 字符串和切片转换
- ...

指针 - unsafe 包的灵魂

Go版无类型指针和数值化的指针:

```
var p unsafe.Pointer = nil         // unsafe
var q uintptr        = uintptr(p) // builtin
```

C版无类型指针和数值化的指针:

```
void      *p = NULL;
uintptr_t q = (uintptr_t)(p); // <stdint.h>
```

- unsafe.Pointer 是 Go指针 和 C指针 转换的中介
- uintptr 是 Go 中 数值 和 指针 转换的中介

unsafe 包

```
type ArbitraryType int
type Pointer *ArbitraryType

func Sizeof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr

func Offsetof(x ArbitraryType) uintptr
```

- Pointer: 面向编译器无法保证安全的指针类型转换
- Sizeof: 值所对应变量在内存中的大小
- Alignof: 值所对应变量在内存中地址几个字节对齐
- Offsetof: 结构体中成员的偏移量

unsafe 包 - C语言版本

```
typedef void* Pointer;

sizeof(type or expression); // C
offsetof(type, member);     // <stddef.h>
alignof(type-id);           // C++ 11
```

- C指针的安全性永远需要自己负责
- sizeof 是关键字, 语义和 Go 基本一致
- offsetof 是宏, 展开为表达式, 语义和 Go 基本一致
- alignof 是新特性, 可忽略

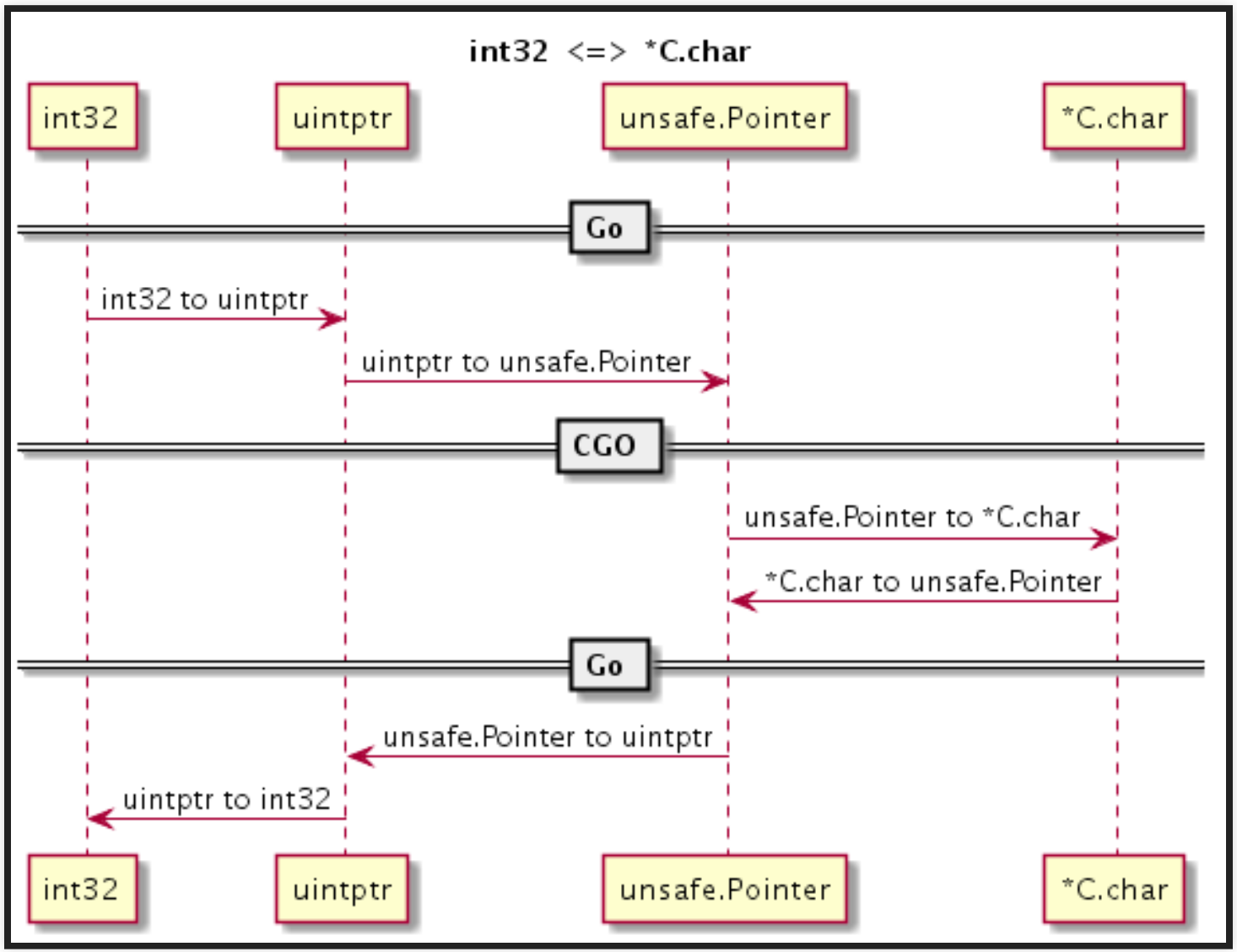
Go字符串和切片的结构

```
type reflect.StringHeader struct {
    Data uintptr
    Len  int
}
type reflect.SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

```
typedef struct { const char *p; GoInt n; } GoString;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;
```

- reflect 包定义的结构和CGO生成的C结构是一致的
- GoString 和 GoSlice 和头部结构是兼容的

实战: int32 和 *C.char 相互转换(A)

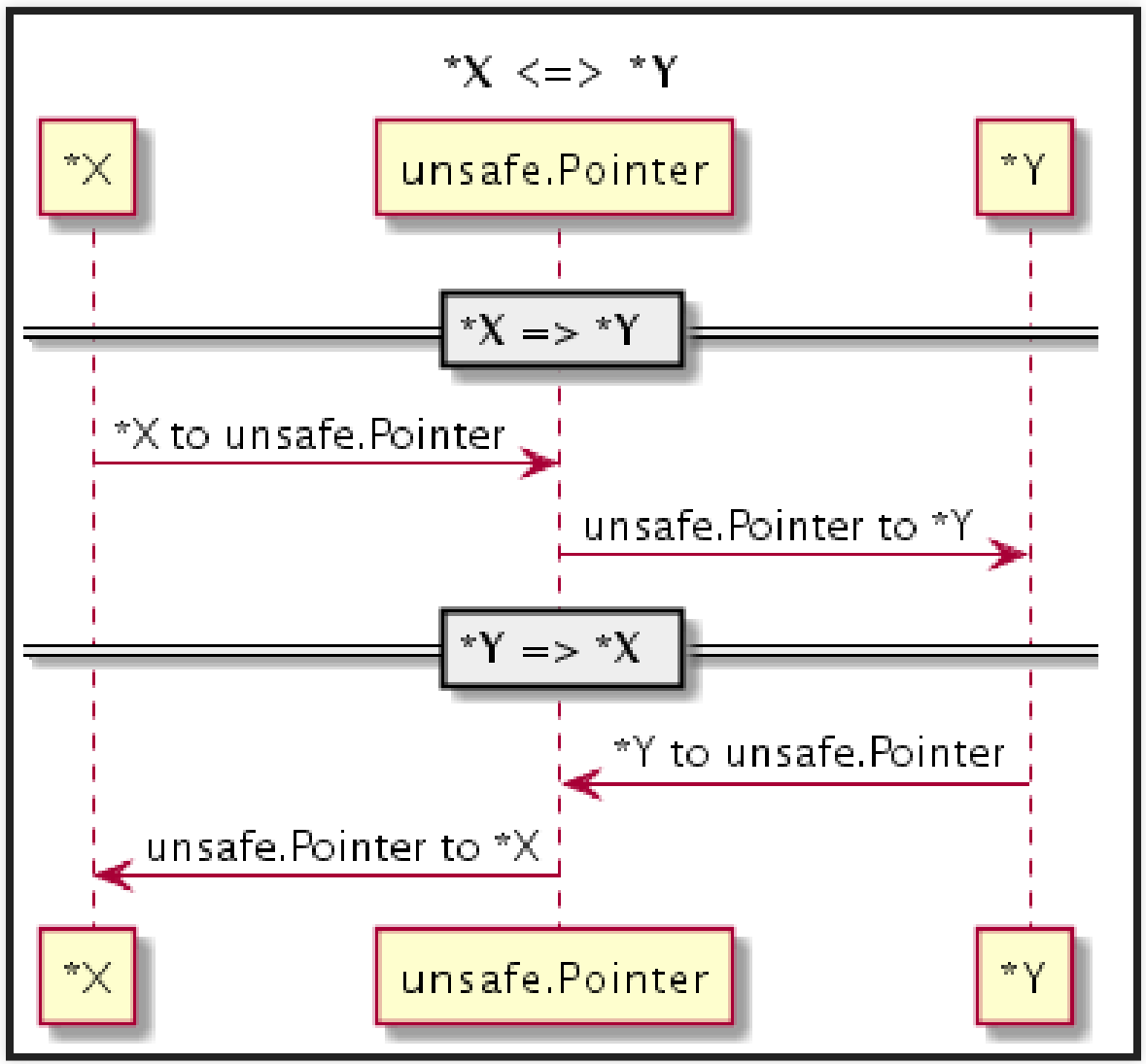


实战: int32 和 *C.char 相互转换(B)

```
// int32 => *C.char
var x = int32(9527)
var p *C.char = (*C.char)(unsafe.Pointer(uintptr(x)))

// *C.char => int32
var y *C.char
var q int32 = int32(uintptr(unsafe.Pointer(y)))
```

1. 第一步: int32 => uintprt
2. 第二步: uintptr => unsafe.Pointer
3. 第三步: unsafe.Pointer => *C.char
4. 反之亦然

实战: *x 和 *y 相互转换(A)

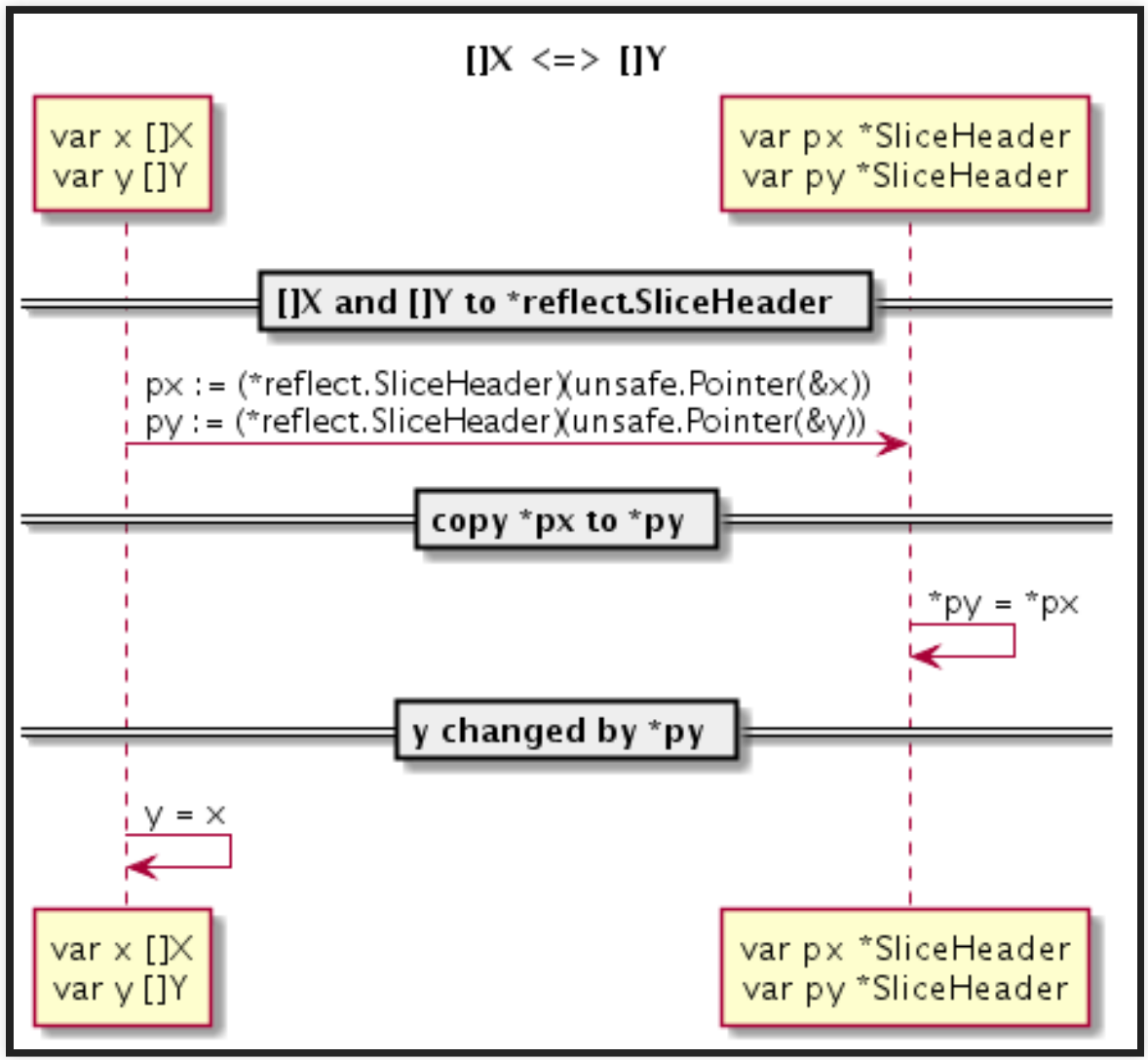

*X <=> *Y
*X
unsafe.Pointer
*Y
*X => *Y
*X to unsafe.Pointer
unsafe.Pointer to *Y
*Y => *X
*Y to unsafe.Pointer
unsafe.Pointer to *X
*X
unsafe.Pointer
*Y

实战: *X 和 *Y 相互转换(B)

```
var p *X
var q *Y

q = (*Y)(unsafe.Pointer(p)) // *X => *Y
p = (*X)(unsafe.Pointer(q)) // *Y => *X
```

1. 第一步: *X => unsafe.Pointer
2. 第二步: unsafe.Pointer => *Y
3. 反之亦然

实战: [ ]X 和 [ ]Y 相互转换(A)


[]X <=> []Y
var x []X
var y []Y
var px *SliceHeader
var py *SliceHeader
[]X and []Y to *reflect.SliceHeader
px := (*reflect.SliceHeader)(unsafe.Pointer(&x))
py := (*reflect.SliceHeader)(unsafe.Pointer(&y))
copy *px to *py
*py = *px
y changed by *py
y = x
var x []X
var y []Y
var px *SliceHeader
var py *SliceHeader

实战: []X 和 []Y 相互转换(B)

```
var p []X
var q []Y // q = p

pHdr := (*reflect.SliceHeader)(unsafe.Pointer(&p))
qHdr := (*reflect.SliceHeader)(unsafe.Pointer(&q))

pHdr.Data = qHdr.Data
pHdr.Len = qHdr.Len * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0]
pHdr.Cap = qHdr.Cap * unsafe.Sizeof(q[0]) / unsafe.Sizeof(p[0]
```

- 所有切片拥有相同的头部 `reflect.SliceHeader`
- 重新构造切片头部即可完成转换

示例: float64 数组排序优化

```
func main() {
    // []float64 强制类型转换为 []int
    var a = []float64{4, 2, 5, 7, 2, 1, 88, 1}
    var b []int = ((*[1 << 20]int)(unsafe.Pointer(&a[0])))[:le

    // 以int方式给float64排序
    sort.Ints(b)
}
```

- float64遵循IEEE754浮点数标准特性
- 当浮点数有序时对应的整数也必然是有序的

函数调用

- Go调用C函数
- C调用Go导出函数
- 深度调用: Go => C => Go => C

Go调用C函数(A)

```
/*
static int add(int a, int b) {
    return a+b;
}
*/
import "C"

func main() {
    C.add(1, 1)
}
```

- C.add 通过的 C 虚拟包访问
- 最终会转为 _Cfunc_add 名字

Go调用C函数(B)

```
/*
static int add(int a, int b) {
    return a+b;
}
*/
import "C"
import "fmt"

func main() {
    v, err := C.add(1, 1)
    fmt.Println(v, err)

    // Output:
    // 4 <nil>
}
```

- 任何C函数都可以带2个返回值
- 第二个返回值是 errno, 对应 error 接口类型

Go调用C函数(C)

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

static void seterrno(int v) {
    errno = v;
}
*/
import "C"
import "fmt"

func main() {
    _, err = C.seterrno(9527)
    fmt.Println(err)

    // Output:
```

Go调用C函数(D)

```
// static void noreturn() {}
import "C"
import "fmt"

func main() {
    x, _ := C.noreturn()
    fmt.Printf("%#v\n", x)

    // Output:
    // main._Ctype_void{}
}
```

- 甚至可以获取一个 void 类型函数的返回值
- 返回值类型: `type _Ctype_void [0]byte`

导出Go函数(A)

main.go:

```
import "C"

//export GoAdd
func GoAdd(a, b C.int) C.int {
    return a+b
}
```

- 可以导出私有函数
- 导出C函数名没有名字空间约束, 需保证全局没有重名
- main 包的导出函数会在 `_cgo_export.h` 声明

导出Go函数(B)

add.h:

```
int c_add(int a, int b);
```

add.c:

```
#include "add.h"
#include "_cgo_export.h"

int c_add(int a, int b) {
    return GoAdd(a, b)
}
```

- 在C文件中使用 _cgo_export.h 头文件
- C文件必须在同一个包, 否则会找不到头文件

导出Go函数(C)

main.go:

```
//export int GoAdd(int a, int b);
//#include "add.h"
import "C"

func main() {
    C.GoAdd(1, 1)
    C.c_add(2, 2)
}
```

导出Go函数(D)

```
// extern void SayHello(GoString s); // GoString 在哪定义?
import "C"

//export SayHello
func SayHello(s string) {
    fmt.Print(s)
}
```

- 导出函数的参数是Go字符串
- C类型为 GoString, 在 _cgo_export.h 文件定义
- 要使用 GoString 类型就要引用 _cgo_export.h 文件
- 这时候该如何手写 SayHello 函数的声明?

导出Go函数(E)

```
// +build go1.10

// extern void SayHello(_GoString_ s);
import "C"

//export SayHello
func SayHello(s string) {
    fmt.Print(s)
}
```

- Go 1.10 增加了 _GoString_ 类型
- _GoString_ 是预定义的类型, 和 GoString 等价
- 避免手写函数声明时出现循环依赖

深度调用: Go => C => Go => C (A)

```
/*
static int c_add(int a, int b) {
    return a+b;
}

static int go_add_proxy(int a, int b) {
    extern int GoAdd(int a, int b);
    return GoAdd(a, b);
}
*/
import "C"
```

- go_add_proxy 调用Go导出的 GoAdd

深度调用: Go => C => Go => C (B)

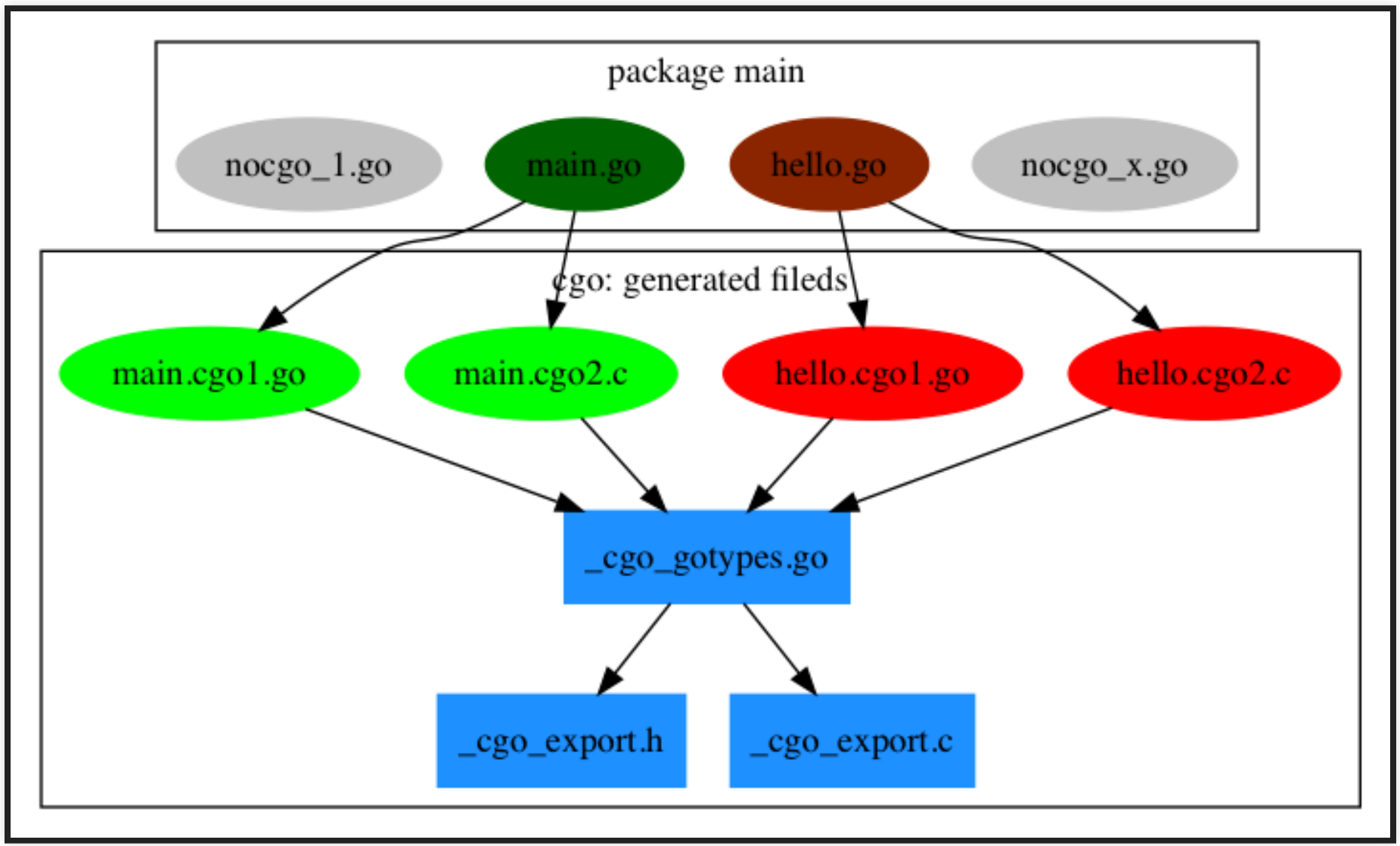
```
func main() {
    C.c_add(1, 1)
}

//export GoAdd
func GoAdd(a, b C.int) C.int {
    return a + b
}
```

- Go:main => C:go_add_proxy => Go:GoAdd

CGO内部机制

- CGO生成的中间文件
- 内部调用流程: Go -> C
- 内部调用流程: C -> Go

CGO生成的中间文件

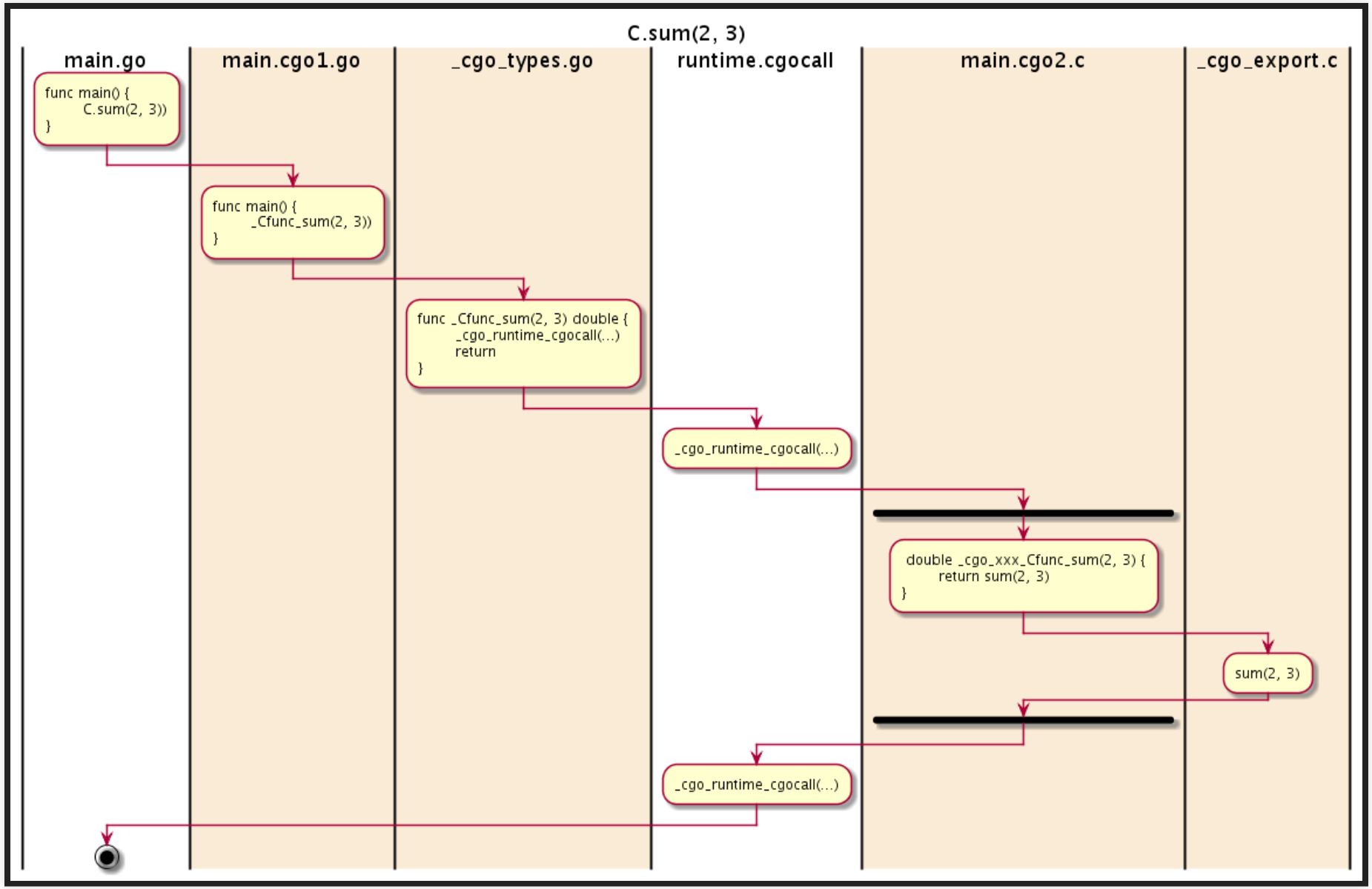

package main
nocgo_1.go
main.go
hello.go
nocgo_x.go
cgo: generated fileds
main.cgo1.go
main.cgo2.c
hello.cgo1.go
hello.cgo2.c
_cgo_gotypes.go
_cgo_export.h
_cgo_export.c

内部调用流程: Go -> C

```go
package main

//int sum(int a, int b);
import "C"

func main() {
    C.sum(1, 2)
}
```

1. `C.sum => _Cfunc_sum`
2. `runtime.cgocall`
3. newthread: `sum`

内部调用流程: Go -> C


C.sum(2, 3)
main.go
main.cgo1.go
_cgo_types.go
runtime.cgocall
main.cgo2.c
_cgo_export.c
func main() {
C.sum(2, 3))
}
func main() {
_Cfunc_sum(2, 3))
}
func _Cfunc_sum(2, 3) double {
_cgo_runtime_cgocall(...)
return
}
_cgo_runtime_cgocall(...)
double _cgo_xxx_Cfunc_sum(2, 3) {
return sum(2, 3)
}
sum(2, 3)
_cgo_runtime_cgocall(...)

内部调用流程: C -> Go

sum.go

```
//int sum(int a, int b);
import "C"

//export sum
func sum(a, b C.int) C.int {
    return a + b
}
```

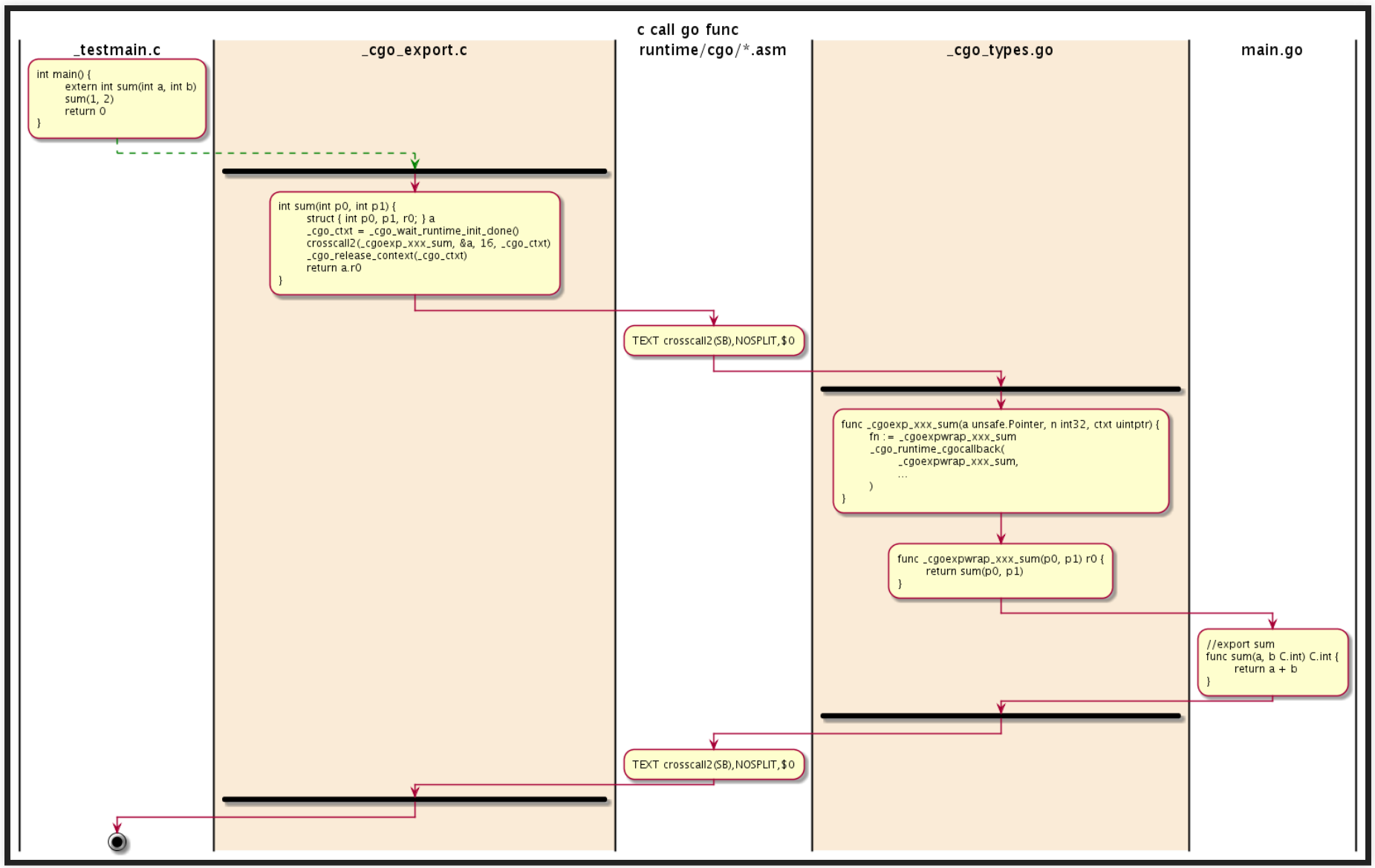
main.c:

```
int main() {
    extern int sum(int a, int b);
    sum(1, 2);
    return 0;
}
```

内部调用流程: C -> Go

1. c thread: `sum`
2. `ctx = cgo_runtime_init_done()`
3. `runtime/cgo/crosscall2`
4. goroutine: `_cgoexp_xxx_sum`
5. goroutine: `_cgowrap_xxx_sim`
6. goroutine: `sum`
7. `cgo_release_context(ctx)`

内部调用流程: C -> Go


c call go func
_testmain.c
_cgo_export.c
runtime/cgo/*.asm
_cgo_types.go
main.go
int main() {
extern int sum(int a, int b)
sum(1, 2)
return 0
}
int sum(int p0, int p1) {
struct { int p0, p1, r0; } a
_cgo_ctxt = _cgo_wait_runtime_init_done()
crosscall2(_cgoexp_xxx_sum, &a, 16, _cgo_ctxt)
_cgo_release_context(_cgo_ctxt)
return a.r0
}
TEXT crosscall2(SB),NOSPLIT,$0
func _cgoexp_xxx_sum(a unsafe.Pointer, n int32, ctxt uintptr) {
fn := _cgoexpwrap_xxx_sum
_cgo_runtime_cgocallback(
_cgoexpwrap_xxx_sum,
...
)
}
func _cgoexpwrap_xxx_sum(p0, p1) r0 {
return sum(p0, p1)
}
//export sum
func sum(a, b C.int) C.int {
return a + b
}
TEXT crosscall2(SB),NOSPLIT,$0

C.xxx 类型不能跨越多个包

- 因为 C.xxx 最终对应 _Ctype_xxx 内部类型
- 因此不同包之间的 C.int 并不是相同的类型

实战: 包装 C.qsort

```
#include <stdlib.h>

void qsort(
    void* base, size_t num, size_t size,
    int (*compare)(const void* a, const void* b)
);
```

- qsort 为 C 语言高阶函数
- 通过传入自定义的比较函数进行快排序
- 尝试包装为Go版本的qsort
- 目标: 简单易用, 功能灵活

qsort 包装的分阶段目标

- 第一步: 用于Go固定类型数组的排序
- 第二步: 在Go中自传入比较函数
- 第三步: Go比较函数类型的简化
- 第四步: 适配更多数组类型

C中的qsort

```c
#include <stdlib.h>

#define DIM(x) (sizeof(x)/sizeof((x)[0]))

static int compare(const void* a, const void* b) {
    return ( *(int*)a - *(int*)b );
}

int main() {
    int values[] = { 42, 9, 101, 95, 27, 25 };
    qsort(values, DIM(values), sizeof(values[0]), compare);
    return 0;
}
```

Go中的qsort(A)

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

#define DIM(x) (sizeof(x)/sizeof((x)[0]))

static int compare(const void* a, const void* b) {
    return ( *(int*)a - *(int*)b );
}

static void qsort_proxy(int values[], size_t len, size_t elems
    qsort(values, DIM(values), sizeof(values[0]), compare);
}
*/
import "C"
```

- qsort_proxy 为代理函数
- 不能改变比较函数

Go中的qsort(B)

```
import "unsafe"
import "fmt"

func main() {
    values := []int32{ 42, 9, 101, 95, 27, 25 };
    C.qsort_proxy(
        unsafe.Pointer(&values[0]),
        C.size_t(len(values)),
        C.size_t(unsafe.Sizeof(values[0])),
    )
    fmt.Println(values)
}
```

- 可用于排序Go数组
- 不能改变比较函数

C比较函数回调Go导出函数

```
/*
extern int go_qsort_compare(void* a, void* b);

static int compare(const void* a, const void* b) {
    return go_qsort_compare((void*)(a), (void*)(b))
}
*/
import "C"

//export go_qsort_compare
func go_qsort_compare(a, b unsafe.Pointer) C.int {
    pa := (*C.int)(a)
    pb := (*C.int)(b)
    return C.int(*pa - *pb)
}
```

- 为何不直接传入 go_qsort_compare ?

直接使用Go导出的比较函数

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

typedef int (*qsort_cmp_func_t)(const void* a, const void* b);
extern int go_qsort_compare(void* a, void* b);
*/
import "C"

func main() {
    values := []int32{42, 9, 101, 95, 27, 25}

    C.qsort(unsafe.Pointer(&values[0]),
        C.size_t(len(values)), C.size_t(unsafe.Sizeof(values[0
        (C.qsort_cmp_func_t)(unsafe.Pointer(C.go_qsort_compare
    )
```

传入闭包比较函数(A)

```
import "C"

//export go_qsort_compare
func go_qsort_compare(a, b unsafe.Pointer) C.int {
    return go_qsort_compare_info.fn(a, b)
}

var go_qsort_compare_info struct {
    fn func(a, b unsafe.Pointer) C.int
    sync.RWMutex
}
```

- go_qsort_compare_info 保存闭包比较函数信息
- 为了并发安全, 需要加锁保护

传入闭包比较函数(B)

```
func main() {
    values := []int32{42, 9, 101, 95, 27, 25}

    go_qsort_compare_info.Lock()
    defer go_qsort_compare_info.Unlock()
    go_qsort_compare_info.fn = func(a, b unsafe.Pointer) C.int
        pa := (*C.int)(a)
        pb := (*C.int)(b)
        return C.int(*pa - *pb)
    }

    C.qsort(unsafe.Pointer(&values[0]),
        C.size_t(len(values)), C.size_t(unsafe.Sizeof(values[0
        (C.qsort_cmp_func_t)(unsafe.Pointer(C.go_qsort_compare
    )
```

- 为了并发安全, 需要加锁保护

传入闭包比较函数(C)

```
func qsort(values []int32, fn func(a, b unsafe.Pointer) C.int)
    go_qsort_compare_info.Lock()
    defer go_qsort_compare_info.Unlock()

    go_qsort_compare_info.fn = fn

    C.qsort(
        unsafe.Pointer(&values[0]),
        C.size_t(len(values)),
        C.size_t(unsafe.Sizeof(values[0])),
        (C.qsort_cmp_func_t)(unsafe.Pointer(C.go_qsort_compare
    )
}
```

- 包装了Go版本的qsort函数, 支持传入闭包比较函数
- 不足: 只支持 `[]int32` 类型数组
- 不足: 比较函数依然难用

通过接口适配更多数组类型

```
func qsort(slice interface{}, fn func(a, b unsafe.Pointer) C.i
    sv := reflect.ValueOf(slice)
    if sv.Kind() != reflect.Slice {
        panic("not slice type")
    }

    go_qsort_compare_info.Lock()
    defer go_qsort_compare_info.Unlock()
    go_qsort_compare_info.fn = fn

    C.qsort(
        unsafe.Pointer(unsafe.Pointer(sv.Index(0).Addr().Point
        C.size_t(sv.Len()), C.size_t(sv.Type().Elem().Size()),
        (C.qsort_cmp_func_t)(unsafe.Pointer(C.go_qsort_compare
    )
```

- 改用空接口接收不同数组类型(没有范型的恶果)
- 通过 reflect 来获取数组或切片的信息

简化比较函数类型(A)

```
func qsort(slice interface{}, fn func(a, b int) int) {
    ...
}
```

- 参考sort包的less函数: `func(i, j int) int`
- 将元素指针转为数组下标, 配合闭包函数更简单
- 返回值转为普通 int 类型

简化比较函数类型(B)

- 如何将比较函数的指针转为数组下标?
- 通过元素指针减去数组开始地址似乎可以

- Go中数组的内存地址可能会移动, 如何处理?
- 在C.qsort函数调用时, 此时Go内存已经锁定

- 如果已经调用C.qsort函数, 如何告诉比较函数地址?
- 可做一个qosrt代理函数, 在入口保存数组地址

- 以上方案似乎可行

简化比较函数类型(C)

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

extern int  go_qsort_compare(void* a, void* b);
extern void go_qsort_compare_save_base(void* base);

static void qsort_proxy(
    void* base, size_t num, size_t size,
    int (*compar)(const void* a, const void* b)
) {
    go_qsort_compare_save_base(base); // 保存数组地址
    qsort(base, num, size, compar);
}
*/
import "C"
```

简化比较函数类型(D)

```
//export go_qsort_compare_save_base
func go_qsort_compare_save_base(base unsafe.Pointer) {
    go_qsort_compare_info.base = uintptr(base)
}

var go_qsort_compare_info struct {
    base     uintptr
    elemsize uintptr
    fn       func(a, b int) int
    sync.RWMutex
}
```

- go_qsort_compare_info 还增加了 elemsize 信息
- elemsize 对应数组元素的大小

简化比较函数类型(E)

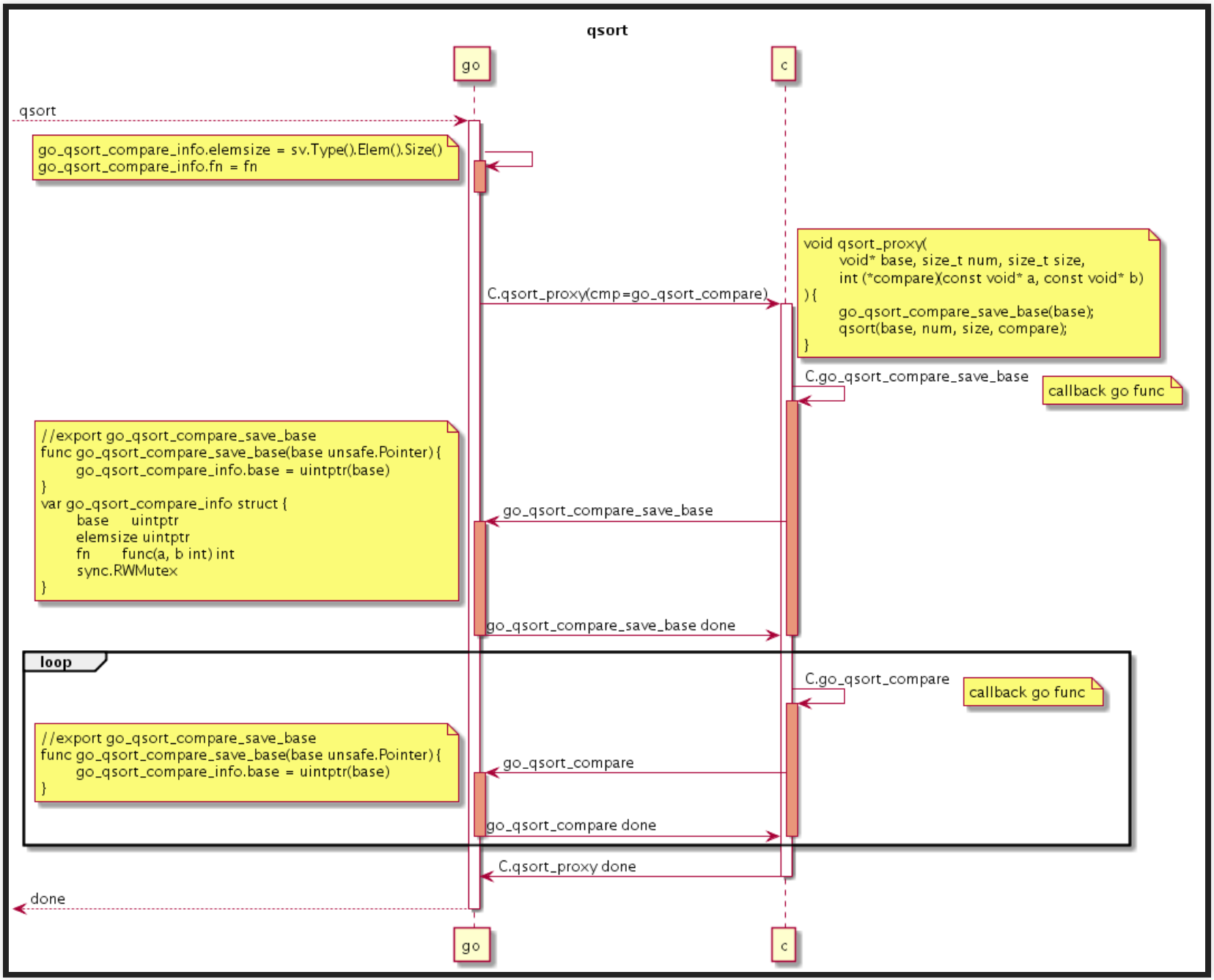
```
//export go_qsort_compare
func go_qsort_compare(a, b unsafe.Pointer) C.int {
    var (
        // array memory is locked
        base     = go_qsort_compare_info.base
        elemsize = go_qsort_compare_info.elemsize
    )

    i := int((uintptr(a) - base) / elemsize)
    j := int((uintptr(b) - base) / elemsize)

    return C.int(go_qsort_compare_info.fn(i, j))
}
```

- 比较函数将指针转为数组的下标
- 然后调用闭包比较函数

qsort最终版本

```
func main() {
    values := []int64{42, 9, 101, 95, 27, 25}

    qsort(values, func(i, j int) int {
        return int(values[i] - values[j])
    })
}

func qsort(slice interface{}, fn func(a, b int) int) {
    ...
}
```

- 闭包的缺点: 需要借助全局变量转为C函数指针
- qsort 对全局资源产生依赖, 对并发有影响


qsort
go
c
qsort
go_qsort_compare_info.elemsize = sv.Type().Elem().Size()
go_qsort_compare_info.fn = fn
void qsort_proxy(
void* base, size_t num, size_t size,
int (*compare)(const void* a, const void* b)
){
go_qsort_compare_save_base(base);
qsort(base, num, size, compare);
}
C.qsort_proxy(cmp=go_qsort_compare)
C.go_qsort_compare_save_base
callback go func
//export go_qsort_compare_save_base
func go_qsort_compare_save_base(base unsafe.Pointer) {
go_qsort_compare_info.base = uintptr(base)
}
var go_qsort_compare_info struct {
base uintptr
elemsize uintptr
fn func(a, b int) int
sync.RWMutex
}
go_qsort_compare_save_base
go_qsort_compare_save_base done
loop
C.go_qsort_compare
callback go func
//export go_qsort_compare_save_base
func go_qsort_compare_save_base(base unsafe.Pointer) {
go_qsort_compare_info.base = uintptr(base)
}
go_qsort_compare
go_qsort_compare done
C.qsort_proxy done
done
go
c

内存模型

- 内存是如何对齐
- 内存是如何布局的(堆/栈)
- GC对内存有何影响
- 动态栈有何影响
- 顺序一致性内存模型

结构体对齐(A)

- CPU对基础类型有对齐要求, 比如int32要求4字节对齐

- 推论1: 数组中的每个元素也要对齐
- 推论2: 结构体中的每个成员也要对齐
- 推论3: 结构体数组中每个元素的每个成员都要对齐

结构体对齐(B)

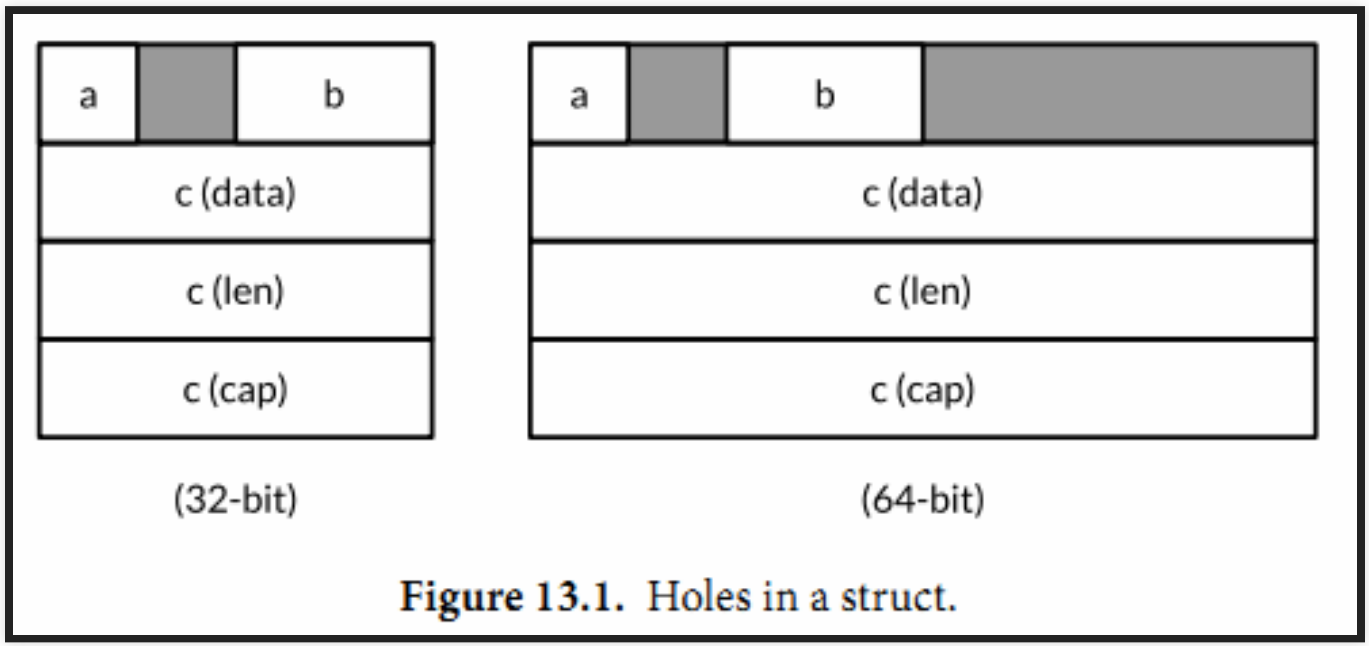


Figure 13.1. Holes in a struct.

堆和栈(A)

- Go中有堆也有栈
- 但是不知道它们在哪里
- 不仅仅如此, 栈还会到处乱窜...

- 推论1: 无法知道变量是在栈上还是在堆上
- 推论2: 任何变量都可能在内存到处乱窜

- 以上对Go语言码农是透明的
- 但 cgo 不是 Go

堆和栈(B)

```
func main() {
    p := intptr(getv()) // p 安全吗?
    v := *(*int)(unsafe.Pointer(p))
    println(v)
}
func getv() *int {
    // x 在对上还是栈上?
    var x = 42
    return &x
}
```

- 此时x应该在堆上
- 原因: getv 返回时, x 地址依然有效, 而栈帧已经被销毁

C 内存到 Go 内存(A)

- C中栈内存不能返回(函数调用返回就被回收)
- C堆中的内存地址是稳定的

- 可以将C堆内存转为任意其它数值类型, 转回依然在

C 内存到 Go 内存(B)

```
type VoidPointer uintptr

func NewVoidPointerFrom(p unsafe.Pointer) VoidPointer {
    return VoidPointer(p)
}

func Malloc(n int) VoidPointer {
    return VoidPointer(C.malloc(C.size_t(n)))
}
func (p VoidPointer) Free() {
    C.free(unsafe.Pointer(p))
}

func (p VoidPointer) ByteSlice(n int) []byte {
    return ((*[1 << 31]byte)(unsafe.Pointer(p)))[0:n:n]
```

Go内存临时到C内存(A)

- cgo规定: Go函数返回指针不能包含Go内存
- cgo规定: 调用C函数, 可以传入Go内存, 返回失效

- 问题: 调用C函数, 回调Go函数返回Go内存, 如何?

Go内存临时到C内存(B) - 01

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

extern int* go_get_arr_ptr_v1(int *arr, int idx);

static void print_array_v1(int *arr, int n) {
    int i;
    for(i = 0; i < n; i++) {
        int *p = (int*)(go_get_arr_ptr_v1(arr, i));
        printf("%d ", *p);
    }
    printf("\n");
}
*/
```

- go_get_arr_ptr_v1 Go实现, 可能返回Go内存

Go内存临时到C内存(B) - 02

```
func main() {
    values := []int32{42, 9, 101, 95}

    // panic: runtime error: cgo result has Go pointer
    C.print_array_v1(
        (*C.int)(unsafe.Pointer(&values[0])),
        C.int(len(values)),
    )
}

//export go_get_arr_ptr_v1
func go_get_arr_ptr_v1(arr *C.int, idx C.int) *C.int {
    base := uintptr(unsafe.Pointer(arr))
    p := (*C.int)(unsafe.Pointer(base + uintptr(idx)*4))
    return p
```

- go run main.go
- GODEBUG=cgocheck=0 go run main.go

Go内存临时到C内存(C)

```
//export go_get_arr_ptr_v2
func go_get_arr_ptr_v2(arr *C.int, idx C.int) C.uintptr_t {
    base := C.uintptr_t(uintptr(unsafe.Pointer(arr)))
    p := base + C.uintptr_t(idx)*4
    return p
}
```

- 返回值避免用指针类型, 避免cgo检测指针
- 这是回调上下文, 返回的Go内存被锁定了

C长期持有Go内存(A)

```
type ObjectId int32

var refs struct {
    sync.Mutex
    objs map[ObjectId]interface{}
    next ObjectId
}

func init() {
    refs.Lock()
    defer refs.Unlock()

    refs.objs = make(map[ObjectId]interface{})
    refs.next = 1000
}
```

- 将Go内存对象映射为int类型的id
- id是稳定的, 不会发生变化

C长期持有Go内存(B)

```
func NewObjectId(obj interface{}) ObjectId {
    refs.Lock()
    defer refs.Unlock()

    id := refs.next
    refs.next++

    refs.objs[id] = obj
    return id
}
func (id ObjectId) Get() interface{} {
    refs.Lock()
    defer refs.Unlock()

    return refs.objs[id]
```

- 可以从id解包出真实的Go对象

C长期持有Go内存(C)

```
func (id *ObjectId) Free() interface{} {
    refs.Lock()
    defer refs.Unlock()

    obj := refs.objs[*id]
    delete(refs.objs, *id)
    *id = 0

    return obj
}
```

- 不需要时需要释放id, 避免资源泄露

C长期持有Go内存(D)

```
//export char* NewGoString(char* s);
//export void FreeGoString(char* s);
import "C"

//export NewGoString
func NewGoString(s *C.char) *C.char {
    gs := C.GoString(s)
    id := NewObjectId(gs)
    return (*C.char)(unsafe.Pointer(uintptr(id)))
}

//export FreeGoString
func FreeGoString(p *C.char) {
    id := ObjectId(uintptr(unsafe.Pointer(p)))
    id.Free()
```

- id是数值类型, 可以当一个伪指针使用
- 类比: C++中的this也可以当作id

Go访问C++对象

1. 准备一个C++类
2. C++类转C接口
3. C接口函数到Go接口函数
4. Go接口函数到Go对象

Go访问C++对象: 准备一个C++类

```
struct MyBuffer {
    std::string* s_;

    MyBuffer(int size) {
        this->s_ = new std::string(size, char('\0'));
    }
    ~MyBuffer() {
        delete this->s_;
    }
    int Size() const {
        return this->s_->size();
    }
    char* Data() {
        return (char*)this->s_->data();
    }
```

- 值类型风格的对象(并不推荐)
- 推荐 new/delete 风格

Go访问C++对象: C++中的使用方式

```
int main() {
    auto pBuf = new MyBuffer(1024);

    auto data = pBuf->Data();
    auto size = pBuf->Size();

    delete pBuf;
}
```

- 通过 new 创建对象, 避免以值或引用的方式使用对象

Go访问C++对象: 想象为C风格接口

```
int main() {
    MyBuffer* pBuf = NewMyBuffer(1024);

    char* data = MyBuffer_Data(pBuf);
    auto size = MyBuffer_Size(pBuf);

    DeleteMyBuffer(pBuf);
}
```

- new 关键字很容易转为 new 函数
- Go 中 new 也是一个范型函数

Go访问C++对象: C接口

```
// my_buffer_capi.h
typedef struct MyBuffer_T MyBuffer_T;

MyBuffer_T* NewMyBuffer(int size);
void DeleteMyBuffer(MyBuffer_T* p);

char* MyBuffer_Data(MyBuffer_T* p);
int MyBuffer_Size(MyBuffer_T* p);
```

- MyBuffer_T 是一种匿名的结构
- 避免依赖 new/delete 关键字
- 一切都是 C 函数风格

Go访问C++对象: C接口实现(01)

```
#include "./my_buffer.h"

extern "C" {
    #include "./my_buffer_capi.h"
}

struct MyBuffer_T: MyBuffer {
    MyBuffer_T(int size): MyBuffer(size) {}
    ~MyBuffer_T() {}
};

MyBuffer_T* NewMyBuffer(int size) {
    auto p = new MyBuffer_T(size);
    return p;
}
```

- 对外, MyBuffer_T 是一种匿名的结构
- 对内, MyBuffer_T 是一个普通的 C++ 类, 有基类

Go访问C++对象: C接口实现(02)

```
void DeleteMyBuffer(MyBuffer_T* p) {
    delete p;
}

char* MyBuffer_Data(MyBuffer_T* p) {
    return p->Data();
}
int MyBuffer_Size(MyBuffer_T* p) {
    return p->Size();
}
```

- 将类函数转为全局的C函数
- p 对应 this

Go访问C++对象: C函数到Go函数(01)

```
//#include "my_buffer_capi.h"
import "C"

type cgo_MyBuffer_T C.MyBuffer_T

func cgo_NewMyBuffer(size int) *cgo_MyBuffer_T {
    p := C.NewMyBuffer(C.int(size))
    return (*cgo_MyBuffer_T)(p)
}

func cgo_DeleteMyBuffer(p *cgo_MyBuffer_T) {
    C.DeleteMyBuffer((*C.MyBuffer_T)(p))
}
```

- 只是为了便于理解, 真实环节可以省略这层封装
- 这是CGO桥接两个语言的关键部分

Go访问C++对象: C函数到Go函数(02)

```
func cgo_MyBuffer_Data(p *cgo_MyBuffer_T) *C.char {
    return C.MyBuffer_Data((*C.MyBuffer_T)(p))
}

func cgo_MyBuffer_Size(p *cgo_MyBuffer_T) C.int {
    return C.MyBuffer_Size((*C.MyBuffer_T)(p))
}
```

Go访问C++对象: 包装为Go对象(01)

```
type MyBuffer struct {
    cptr *cgo_MyBuffer_T
}

func NewMyBuffer(size int) *MyBuffer {
    return &MyBuffer{
        cptr: cgo_NewMyBuffer(size),
    }
}

func (p *MyBuffer) Delete() {
    cgo_DeleteMyBuffer(p.cptr)
}
```

- 现在已经完全是 Go 语言的问题了

Go访问C++对象: 包装为Go对象(02)

```
func (p *MyBuffer) Data() []byte {
    data := cgo_MyBuffer_Data(p.cptr)
    size := cgo_MyBuffer_Size(p.cptr)
    return ((*[1 << 31]byte)(unsafe.Pointer(data)))[0:int(size
}
```

```
func main() {
    buf := NewMyBuffer(1024)
    defer buf.Delete()

    copy(buf.Data(), []byte("hello\x00"))
    C.puts((*C.char)(unsafe.Pointer(&(buf.Data()[0]))))
}
```

- 切片包含了地址和长度, 两个方法合一
- C 字符串需要 '\0' 结尾

Go对象导出为C++对象

1. 准备一个Go对象
2. Go对象映射为一个id
3. Go对象对应的id到Go接口函数
4. Go接口函数到C接口函数
5. C接口函数到C++类
6. id就是this指针

准备一个Go对象

```
type Person struct {
    name string
    age  int
}

func NewPerson(name string, age int) *Person {
    return &Person{name: name, age: age}
}

func (p *Person) Set(name string, age int) {
    p.name, p.age = name, age
}

func (p *Person) Get() (name string, age int) {
    return p.name, p.age
```

- 完全是普通的 Go 对象

对应的C接口

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

typedef uintptr_t person_handle_t;

person_handle_t person_new(char* name, int age);
void person_delete(person_handle_t p);

void person_set(person_handle_t p, char* name, int age);
char* person_get_name(person_handle_t p, char* buf, int size);
int person_get_age(person_handle_t p);
```

- 以 C 接口的方式重新抽象前面的 Go 对象
- person_handle_t 类似对象指针, 只是类似
- uintptr_t 类似指针, 但不是指针

Go实现C接口函数(A)

```
//export person_new
func person_new(name *C.char, age C.int) C.person_handle_t {
    id := NewObjectId(NewPerson(C.GoString(name), int(age)))
    return C.person_handle_t(id)
}

//export person_delete
func person_delete(h C.person_handle_t) {
    ObjectId(h).Free()
}

//export person_set
func person_set(h C.person_handle_t, name *C.char, age C.int)
    p := ObjectId(h).Get().(*Person)
    p.Set(C.GoString(name), int(age))
```

- Go对象指针如何转为 person_handle_t ?

Go实现C接口函数(B)

```
//export person_get_name
func person_get_name(h C.person_handle_t, buf *C.char, size C.
    p := ObjectId(h).Get().(*Person)
    name, _ := p.Get()

    n := int(size) - 1
    bufSlice := ((*[1 << 31]byte)(unsafe.Pointer(buf)))[0:n:n]
    n = copy(bufSlice, []byte(name))
    bufSlice[n] = 0

    return buf
}
```

- person_handle_t 如何还原 Go 对象指针？

Go实现C接口函数(C)

```
//export person_get_age
func person_get_age(h C.person_handle_t) C.int {
    p := ObjectId(h).Get().(*Person)
    _, age := p.Get()
    return C.int(age)
}
```

- Go 对象指针移动了会如何？

C接口到C++类(A)

```
extern "C" {
    #include "./person_capi.h"
}

struct Person {
    person_handle_t goobj_;

    Person(const char* name, int age) {
        this->goobj_ = person_new((char*)name, age);
    }
    ~Person() {
        person_delete(this->goobj_);
    }
```

- C++ 的槽点: 每次包装时都需要一层内存隔离

C接口到C++类(B)

```
    void Set(char* name, int age) {
        person_set(this->goobj_, name, age);
    }
    char* GetName(char* buf, int size) {
        return person_get_name(this->goobj_ buf, size);
    }
    int GetAge() {
        return person_get_age(this->goobj_);
    }
}
```

- 这是各种 C++ 教程推荐的技术
- 但是我不喜欢!

C接口到C++类(C)

```
int main() {
    auto p = new Person("gopher", 10);

    char buf[64];
    char* name = p->GetName(buf, sizeof(buf)-1);
    int age = p->GetAge();

    printf("%s, %d years old.\n", name, age);
    delete p;

    return 0;
}
```

id就是this

```
truct Person {
    static Person* New(const char* name, int age) {
        return (Person*)person_new((char*)name, age);
    }
    void Delete() {
        person_delete(person_handle_t(this));
    }

    void Set(char* name, int age) {
        person_set(person_handle_t(this), name, age);
    }
    char* GetName(char* buf, int size) {
        return person_get_name(person_handle_t(this), buf, siz
    }
};
```

- C++ 中的 this 是什么 (用Go的思维)?
- this 就是一个函数参数
- class 是否也是必须的?

还我自由的指针(Go)

```
type Int int

func (p Int) Twice() int {
    return int(p)*2
}
```

```
func main() {
    var x = int(42)
    fmt.Println(Int(x).Twice())
}
```

- 不需要额外的类包装, 仅是类型转换就可扩展方法
- Go 的方法是绑定到类型的

还我自由的指针(C++)

```
struct Int {
    int Twice() {
        const int* p = (int*)(this);
        return (*p) * 2;
    }
};
```

```
int main() {
    int x = 42;
    int v = ((Int*)(&x))->Twice();
    printf("%d\n", v);
    return 0;
}
```

- C++ 的方法其实也可以用于普通非 class 类型
- C++ 到普通成员函数其实也是绑定到类型的
- 只有纯虚方法是绑定到对象, 那就是接口

静态库和动态库

- 如何使用静态库
- 如何使用动态库
- 如何导出静态库
- 如何导出动态库
- 动态库的风险

如何使用静态库(A)

```
// number/number.h

int number_add_mod(int a, int b, int mod);
```

```
// number/number.c
int number_add_mod(int a, int b, int mod) {
    return (a+b)%mod;
}
```

```
$ cd ./number
$ gcc -c -o number.o number.c
$ ar rcs libnumber.a number.o
```

- cgo最终使用gcc的ld链接命令
- 只要是gcc兼容的`libxxx.a`格式的动态库均可使用

如何使用静态库(B)

```
//#cgo CFLAGS: -I./number
//#cgo LDFLAGS: -L${SRCDIR}/number -lnumber
//
//#include "number.h"
import "C"
import "fmt"

func main() {
    fmt.Println(C.number_add_mod(10, 5, 12))
}
```

- -lnumber对应链接libnumber.a库
- -L${SRCDIR}/number指定链接库所在目录
- -L参数必须绝对路径(ld历史遗留问题导致)

如何使用动态库 - so 格式

```
// number/number.h

int number_add_mod(int a, int b, int mod);
```

```
// number/number.c
int number_add_mod(int a, int b, int mod) {
    return (a+b)%mod;
}
```

```
$ cd ./number
$ gcc -shared -o libnumber.so number.c
```

- 链接时 `libnumber.so` 和 `libnumber.a` 等价
- macOS 需要设置 `DYLD_LIBRARY_PATH` 环境变量
- Linux 需要设置 `LD_LIBRARY_PATH` 环境变量

如何使用动态库 - dll 格式

```
; number/number.def
LIBRARY number.dll

EXPORTS
number_add_mod
```

```
$ cl /c number.c
$ link /DLL /OUT:number.dll number.obj number.def
$ dlltool -dllname number.dll --def number.def --output-lib li
```

- MinGW 自带的 dlltool 工具可从 dll 生成 `libxxx.a` 文件
- def 用于控制 dll 导出符号, 也用于生成 `libxxx.a` 文件

如何导出静态库

```
package main

import "C"

func main() {}

//export number_add_mod
func number_add_mod(a, b, mod C.int) C.int {
    return (a + b) % mod
}
```

```
$ go build -buildmode=c-archive -o number.a
```

- cgo还会生成一个 number.h 文件

如何导出动态库(A)

```
$ go build -buildmode=c-shared -o number.so
```

- 动态库和静态库的生成方式类似
- Go1.9 之前不支持 Windows
- Windows 下可从静态库手工生成动态库

如何导出动态库(B)

```
go build -buildmode=c-archive -o number.a
gcc -m64 -shared -o number.dll number.def number.a
lib /def:number.def /machine:x64
```

- 先生成静态库 number.a
- 基于 number.a 手工生成 dll
- VC 的 lib 命令从 dll生成 number.lib 文件

动态库的风险

- Linux 下共享一个 libc.so, 问题不大
- Windows下有多个 msvcrt.dll, 问题非常严重
- Windows下gcc和vc有各自的实现

- 比如从VC6实现的dll中 malloc 了一块内存
- 然后在cgo中 `C.free` 内存, 将导致崩溃

- cgo 打开了一个文件指针 `*C.FILE`
- 然后传入 VC 实现的动态库的 `C.fclose(f)` 关闭
- 因为 `C.FILE` 在 vc 和 gcc 中实现是不同的, 导致崩溃

编写Python扩展

- ctypes: 基于纯C接口
- 基于Py扩展接口

ctypes: 基于C接口(A)

```
// main.go
package main

import "C"
import "fmt"

func main() {}

//export SayHello
func SayHello(name *C.char) {
    fmt.Printf("hello %s!\n", C.GoString(name))
}
```

```
go build -buildmode=c-shared -o say-hello.so main.go
```

- 生成纯C接口的动态库

ctypes: 基于C接口(B)

```
# hello.py
import ctypes

libso = ctypes.CDLL("./say-hello.so")

SayHello = libso.SayHello
SayHello.argtypes = [ctypes.c_char_p]
SayHello.restype = None

SayHello(ctypes.c_char_p(b"hello"))
```

```
$ python3 hello.py
```

- Python3 字节字符串 b"hello"
- 为了便于使用, 需要 py 二次包装

基于Py扩展接口(A)

```
/*
static PyObject* cgo_PyInit_gopkg(void) {
    static PyMethodDef methods[] = {
        {"sum", Py_gopkg_sum, METH_VARARGS, "Add two numbers."
        {NULL, NULL, 0, NULL},
    };
    static struct PyModuleDef module = {
        PyModuleDef_HEAD_INIT, "gopkg", NULL, -1, methods,
    };
    return PyModule_Create(&module);
}
*/
import "C"
```

- 涉及内存的部分必须在C语言定义
- 模块的名字是 gopkg

基于Py扩展接口(B)

```
//export PyInit_gopkg
func PyInit_gopkg() *C.PyObject {
    return C.cgo_PyInit_gopkg()
}

//export Py_gopkg_sum
func Py_gopkg_sum(self, args *C.PyObject) *C.PyObject {
    var a, b C.int
    if C.cgo_PyArg_ParseTuple_ii(args, &a, &b) == 0 {
        return nil
    }
    return C.PyLong_FromLong(C.long(a + b))
}
```

```
$ go build -buildmode=c-shared -o gopkg.so main.go
```

- 模块的方法函数可用Go实现
- 生成的动态库名要和模块名一致 gopkg.so

编译和链接参数

- CFLAGS/CPPFLAGS/CXXFLAGS
- LDFLAGS
- pkg-config
- 自定义 pkg-config
- `go get` 链
- 多个非main包中导出C函数

- CGO_XXX_ALLOW 白名单参数(Go1.10)
- CC_FOR_goos_goarch(Go1.10)

CFLAGS/CPPFLAGS/CXXFLAGS

- CFLAGS 只包含纯 C 代码(`*.c`)
- CPPFLAGS 包含 C/C++ 代码
  (`*.c`,`*.cc`,`*.cpp`,`*.cxx`)
- CXXFLAGS 只包含纯 C++ 代码
  (`*.cc`,`*.cpp`,`*.cxx`)

LDFLAGS

- 链接阶段没有C和C++之分
- 链接库的路径必须是绝对路径(ld历史依赖问题)
- cgo 中的 `${SRCDIR}` 为当前目录的绝对路径

pkg-config

- #cgo pkg-config xxx 生成编译和链接参数
- 底层调用 pkg-config xxx --cflags 生成编译参数
- 底层调用 pkg-config xxx --libs 生成链接参数

pkg-config: bc 文件

```
# /usr/local/lib/pkgconfig/xxx.bc
Name: xxx
Cflags:-I/usr/local/include
Libs:-L/usr/local/lib –lxxx2
```

- PKG_CONFIG_PATH 指定查询 bc 文件的目录
- #cgo pkg-config xxx 对应 xxx.bc 文件
- Cflags 对应编译参数
- Libs 对应链接参数

自定义 pkg-config(A)

- `PKG_CONFIG` 环境变量可指定自定义 pkg-config 程序
- 只要处理好 `--cflags` 和 `--libs` 两个参数即可

- macOS 下 pkg-config 不支持 Python3
- macOS 下 python3-config 用于生成 Python3 参数
- macOS 下 python3-config 生成参数和 cgo 不兼容
- 解决办法: 自定义 pkg-config

自定义 pkg-config(B)

```
// py3-config.go
func main() {
    for _, s := range os.Args {
        if s == "--cflags" {
            out, _ := exec.Command("python3-config", "--cflags
            out = bytes.Replace(out, []byte("-arch"), []byte{}
            out = bytes.Replace(out, []byte("i386"), []byte{},
            out = bytes.Replace(out, []byte("x86_64"), []byte{
            fmt.Print(string(out))
            return
        }
        if s == "--libs" {
            out, _ := exec.Command("python3-config", "--ldflag
            fmt.Print(string(out))
            return
```

```
$ go build -o py3-config py3-config.go
$ PKG_CONFIG=./py3-config go build -buildmode=c-shared -o gopk
```

go get 链

- pkgA -> pkgB -> pkgC -> pkgD -> ...
- go get pkgA 会自动 go get 依赖的包
- 如果 go get pkgB 失败将导致链条断裂

go get 链: 链条断裂的原因

- 不支持某些系统, 编译失败
- 依赖 cgo, 用户没有安装 gcc
- 依赖 cgo, 但是依赖的库没有安装
- 依赖 pkg-config, windows 上没有安装
- 依赖 pkg-config, 没有找到对应的 bc 文件
- 依赖 自定义的 pkg-config, 需要额外的配置
- 依赖 swig, 用户没有安装 swig, 或版本不对

- 既然用了cgo, gcc是无法绕过的依赖
- 但是其它部分要做到 **零配置**

go get 链: 零配置

```
// z_libwebp_src_dec_alpha.c
#include "./internal/libwebp/src/dec/alpha.c"
```

```
$ go get github.com/chai2010/webp
```

- 包含全部的C/C++代码, go get 时从头构建
- 在当前包创建代理C/C++文件, 包含真实的源文件
- 避免打破原 C/C++ 库的目录结构

多个非main包中导出C函数

- 官方文档说明导出的Go函数要放main包

- 真实情况是其它包的Go导出函数也是有效的
- 因为导出后就可以当作C函数使用, 所以必须有效

- cgo编程原则: 面向C接口编程, 必须手写头文件

- 小问题: `_cgo_export.h` 只包含main包的导出函数
- 手写头文件, 去掉对 `_cgo_export.h` 文件的依赖

CGO_XXX_ALLOW 白名单参数(Go1.10)

- CGO_CFLAGS_ALLOW/CGO_CXXFLAGS_ALLOW
- CGO_LDFLAGS_ALLOW

- gcc 的 `-fplugin` 可调用外部插件
- go get 时会有安全漏洞
- 白名单采用正则匹配

CC_FOR_goos_goarch(Go1.10)

- cgo 交叉编译时指定gcc命令
- 比如 CC_FOR_darwin_arm64 用于 iOS

更多话题

- SWIG
- Go Mobile
- ...

SWIG

```go
// hello_test.go
func TestSayHello(t *testing.T) {
    SayHello()
}
```

```cpp
// hello.cc
#include <iostream>

void SayHello() {
    std::cout << "Hello, World!" << std::endl;
}
```

```
// hello.swigcxx
%module hello

%inline %{
extern void SayHello();
%}
```

- 至少有一个go文件, 用于 go build 触发 swig 命令

Go Mobile 原理

- golang.org/x/mobile/bind/seq/ref.go
- golang.org/x/mobile/bind/objc/seq_darwin.go
- golang.org/x/mobile/bind/objc/seq_darwin.m

参考资源

https://golang.org/cmd/cgo/

https://blog.golang.org/c-go-cgo

https://github.com/golang/go/wiki/cgo

https://golang.org/src/runtime/cgocall.go

https://golang.org/misc/cgo/test/

Thank you

https://github.com/chai2010

https://chai2010.cn

@青云QingCloud