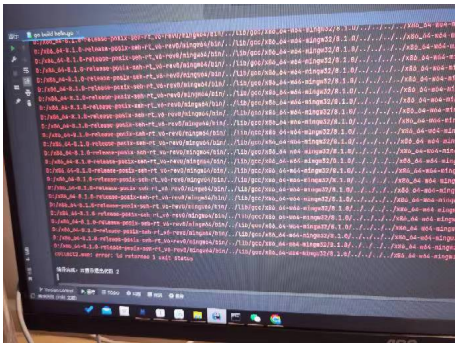
有关GCC环境配置的问题



出现如上图问题 可能是window系统安装了linux gcc

-> uninstall ->install windows gcc MinGW

https://blog.csdn.net/QuantumYou/article/details/119676283?ops_request_misc=&request_id=&biz_id=102&utm_term=windows%20gcc%E4%B8%8B%E8%BD%BD%E5%AE%89%E8%A3%85&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-0-119676283.142^v63^js_top,201^v3^control_1,213^v2^t3_control2&spm=1018.2226.3001.4187

CGO 编程

快速入门

easy demo

```
//hello.go
package main

import "C"

func main() {
    println("hello world!")
    println("it's my first day in HuiYan")
}
```

代码通过`import "C"`语句启用CGO特性，主函数只是通过Go内置的println函数输出字符串，其中并没有任何和CGO相关的代码。虽然没有调用CGO的相关函数，但是`go build`命令会在编译和链接阶段启动gcc编译器，这已经是一个完整的CGO程序了。

string demo

```
//string_demo.go
package main

//you can not put /n between

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

func main() {
    C.puts(C.CString("this is a string from C, demo show!\n"))
}
```

同时包含C语言的 `<stdio.h>` 头文件。然后通过CGO包的 `C.CString` 函数将Go语言字符串转为C语言字符串，最后调用CGO包的 `C.puts` 函数向标准输出窗口打印转换后的C字符串。没有释放使用 `C.CString` 创建的C语言字符串会导致内存泄漏。但是对于这个小程序来说，这样是没有问题的，因为程序退出后操作系统会自动回收程序的所有资源。

self_C demo

```
//self_C.go
package main

/*
#include <stdio.h>

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

func main() {
    C.SayHello(C.CString("this is a C_self demo!\n"))
}
```

同样的，可以将SayHello放在一个以.c后缀的文件中，因为在独立编写的C文件中，为了允许外部使用，需要去掉函数的 static 修饰符

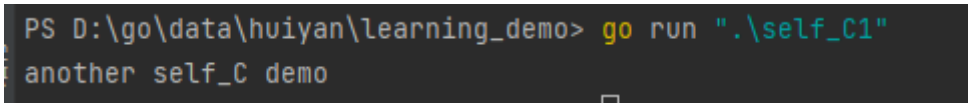
```
#include <stdio.h>

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

然后在CGO部分声明SayHello函数

```
package main

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

func main() {
    C.SayHello(C.CString("another self_C demo\n"))
}
```

注意，如果之前运行的命令是`go run hello.go`或`go build hello.go`的话，此处须使用`go run "your/package"`或`go build "your/package"`才可以。若本就在包路径下的话，也可以直接运行`go run .`或`go build`。

得到结果



module_C demo

定义demo.h的头文件

```
//demo.h
void SayHello(const char* s);
```

实现满足头文件中函数的声明规范，实现对应的module_C.c

```
#include "demo.h"
#include <stdio.h>

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

接口文件demo.h是hello模块的实现者和使用者共同的约定，但是该约定并没有要求必须使用C语言来实现SayHello函数。我们也可以用C++语言来重新实现这个C语言函数：

```
module_C.cpp
#include <iostream>

extern "C" {
    #include "hello.h"
}

void SayHello(const char* s) {
    std::cout << s;
}
```

在C++版本的SayHello函数实现中，我们通过C++特有的`std::cout`输出流输出字符串。不过为了保证C++语言实现的SayHello函数满足C语言头文件demo.h定义的函数规范，我们需要通过`extern "C"`语句指示该函数的链接符号遵循C语言的规则。

go_make_c

先定义相关的接口在 demo.h 的接口文档中

是否定义const 和需求有关 此处结果为不会被修改覆盖

```
//demo.h
void SayHello (/*const*/ char *s);
```

创建一个demo.go

```
//demo.go
package main

import "C"

import "fmt"

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

通过CGO的`//export SayHello`指令将Go语言实现的函数`SayHello`导出为C语言函数。为了适配CGO导出的C语言函数，禁止了在函数的声明语句中的const修饰符。cgo生成的C语言版本SayHello函数最终会通过桥接代码调用Go语言版本的SayHello函数。

通过这样的接口技术，现在将SayHello当作一个标准库进行使用（和puts函数的使用方法类似）

```
//main.go
package main

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

func main() {
    C.SayHello(C.CString("this is go make c function demo\n"))
}
```

最后获得运行结果

```
PS D:\go\data\huiyan\learning_demo> go run ".\go_make_c\"
this is go make c function demo
```

CGO基础

import “C”

```
package main

/*
#include <stdio.h>

void printint(int v){
printf("printint: %d\n",v);
}
*/
import "C"

func main() {
    v := 234
    C.printint(C.int(v))
}
```

```
PS D:\go\data\huiyan\learning_demo\2\import_c> go run demo.go
printint: 234
```

Go是强类型语言，所以cgo中传递的参数类型必须与声明的类型完全一致，而且传递前必须用”C”中的转化函数转换成对应的C类型，不能直接传入Go中类型的变量。同时通过虚拟的C包导入的C语言符号并不需要是大写字母开头，它们不受Go语言的导出规则约束。

cgo将当前包引用的C语言符号都放到了虚拟的C包中，同时当前包依赖的其它Go语言包内部可能也通过cgo引入了相似的虚拟C包，但是不同的Go语言包引入的虚拟的C包之间的类型是不能通用的。这个约束对于要自己构造一些cgo辅助函数时有可能会造成一点的影响。

比如希望在Go中定义一个C语言字符指针对应的CChar类型，然后增加一个GoString方法返回Go语言字符串：

```
package cgo_helper

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

type CChar C.char

func (p *CChar) GoString() string {
    return C.GoString((*C.char)(p))
}

func PrintCString(cs *C.char) {
    C.puts(cs)
}
```

现在我们可能会想在其它的Go语言包中也使用这个辅助函数：

```
package main

//static const char* cs = "hello";
import "C"
import "./cgo_helper"

func main() {
    cgo_helper.PrintCString(C.cs)
}
```

这段代码是不能正常工作的，因为当前main包引入的`C.cs`变量的类型是当前`main`包的cgo构造的虚拟的C包下的`*char`类型（具体点是`*C.char`，更具体点是`*main.C.char`），它和cgo_helper包引入的`*C.char`类型（具体点是`*cgo_helper.C.char`）是不同的。在Go语言中方法是依附于类型存在的，不同Go包中引入的虚拟的C包的类型却是不同的（`main.C`不等`cgo_helper.C`），这导致从它们延伸出来的Go类型也是不同的类型（`*main.C.char`不等`*cgo_helper.C.char`），这最终导致了前面代码不能正常工作。

#cgo语句

在`import "C"`语句前的注释中可以通过`#cgo`语句设置编译阶段和链接阶段的相关参数。编译阶段的参数主要用于定义相关宏和指定头文件检索路径。链接阶段的参数主要是指定库文件检索路径和要链接的库文件。

```
// #cgo CFLAGS: -DPNG_DEBUG=1 -I./include
// #cgo LDFLAGS: -L/usr/local/lib -lpng
// #include <png.h>
import "C"
```

上面的代码中，CFLAGS部分，`-D`部分定义了宏PNG_DEBUG，值为1；`-I`定义了头文件包含的检索目录。LDFLAGS部分，`-L`指定了链接时库文件检索目录，`-l`指定了链接时需要链接png库。

因为C/C++遗留的问题，C头文件检索目录可以是相对目录，但是库文件检索目录则需要绝对路径。在库文件的检索目录中可以通过`${SRCDIR}`变量表示当前包目录的绝对路径：

```
// #cgo LDFLAGS: -L${SRCDIR}/libs -lfoo
```

上面的代码在链接时将被展开为：

```
// #cgo LDFLAGS: -L/go/src/foo/libs -lfoo
```

`#cgo`语句主要影响CFLAGS、CPPFLAGS、CXXFLAGS、FFLAGS和LDFLAGS几个编译器环境变量。LDFLAGS用于设置链接时的参数，除此之外的几个变量用于改变编译阶段的构建参数(CFLAGS用于针对C语言代码设置编译参数)。

其中在windows平台下，编译前会预定义X86宏为1；在非widnows平台下，在链接阶段会要求链接math数学库。这种用法对于在不同平台下只有少数编译选项差异的场景比较适用。

如果在不同的系统下cgo对应着不同的c代码，我们可以先使用`#cgo`指令定义不同的C语言的宏，然后通过宏来区分不同的代码：

```
package main
```

```
/*
#cgo windows CFLAGS: -DCGO_OS_WINDOWS=1
#cgo darwin CFLAGS: -DCGO_OS_DARWIN=1
#cgo linux CFLAGS: -DCGO_OS_LINUX=1

#if defined(CGO_OS_WINDOWS)
    const char* os = "windows";
#elif defined(CGO_OS_DARWIN)
    static const char* os = "darwin";
#elif defined(CGO_OS_LINUX)
    static const char* os = "linux";
#else
#    error(unknown os)
#endif
*/
import "C"

func main() {
    print(C.GoString(C.os))
}
```

成功检测本系统为 windows 系统

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_cgo_demo_go.exe
windows
```

build tag 条件编译

build tag 是在Go或cgo环境下的C/C++文件开头的一种特殊的注释。条件编译类似于前面通过`#cgo`指令针对不同平台定义的宏，只有在对应平台的宏被定义之后才会构建对应的代码。但是通过`#cgo`指令定义宏有个限制，它只能是基于Go语言支持的windows、darwin和linux等已经支持的操作系统。如果我们希望定义一个DEBUG标志的宏，`#cgo`指令就无能为力了。而Go语言提供的build tag 条件编译特性则可以简单做到。

```
// +build debug

package main

var buildMode = "debug"
```

可以用以下命令构建：

```
go build -tags="debug"
go build -tags="windows debug"
```

我们可以通过`-tags`命令行参数同时指定多个build标志，它们之间用空格分隔。

当有多个build tag时，我们将多个标志通过逻辑操作的规则来组合使用。比如以下的构建标志表示只有在"linux/386"或"darwin平台下非cgo环境"才进行构建。

```
// +build linux,386 darwin,!cgo
```

其中`linux,386`中linux和386用逗号链接表示AND的意思；而`linux,386`和`darwin,!cgo`之间通过空白分割来表示OR的意思。

类型转换

数值类型

Go语言中数值类型和C语言数据类型基本上是相似的，以下是它们的对应关系表2-1所示。

C语言类型	CGO类型	Go语言类型
char	C.char	byte
singed char	C.schar	int8
unsigned char	C.uchar	uint8
short	C.short	int16
unsigned short	C.ushort	uint16
int	C.int	int32
unsigned int	C.uint	uint32
long	C.long	int32
unsigned long	C.ulong	uint32
long long int	C.longlong	int64
unsigned long long int	C.ulonglong	uint64
float	C.float	float32
double	C.double	float64
size_t	C.size_t	uint

表 2-1 Go语言和C语言类型对比

需要注意的是，虽然在C语言中`int`、`short`等类型没有明确定义内存大小，但是在CGO中它们的内存大小是确定的。在CGO中，C语言的`int`和`long`类型都是对应4个字节的内存大小，`size_t`类型可以当作Go语言`uint`无符号整数类型对待。

CGO中，虽然C语言的`int`固定为4字节的大小，但是Go语言自己的`int`和`uint`却在32位和64位系统下分别对应4个字节和8个字节大小。如果需要在C语言中访问Go语言的`int`类型，可以通过`GoInt`类型访问，`GoInt`类型在CGO工具生成的`_cgo_export.h`头文件中定义。其实在`_cgo_export.h`头文件中，每个基本的Go数值类型都定义了对应的C语言类型，它们一般都是以单词Go为前缀下面是64位环境下，`_cgo_export.h`头文件生成的Go数值类型的定义，其中`GoInt`和`GoUint`类型分别对应`GoInt64`和`GoUint64`：

```
typedef signed char GoInt8;
typedef unsigned char GoUint8;
typedef short GoInt16;
typedef unsigned short GoUint16;
typedef int GoInt32;
typedef unsigned int GoUint32;
typedef long long GoInt64;
typedef unsigned long long GoUint64;
typedef GoInt64 GoInt;
typedef GoUint64 GoUint;
typedef float GoFloat32;
typedef double GoFloat64;
```

除了`GoInt`和`GoUint`之外，我们并不推荐直接访问`GoInt32`、`GoInt64`等类型。更好的做法是通过C语言的C99标准引入的`<stdint.h>`头文件。为了提高C语言的可移植性，在`<stdint.h>`文件中，不但每个数值类型都提供了明确内存大小，而且和Go语言的类型命名更加一致。Go语言类型`<stdint.h>`头文件类型对比如表2-2所示。

C语言类型	CGO类型	Go语言类型
int8_t	C.int8_t	int8
uint8_t	C.uint8_t	uint8
int16_t	C.int16_t	int16
uint16_t	C.uint16_t	uint16
int32_t	C.int32_t	int32
uint32_t	C.uint32_t	uint32
int64_t	C.int64_t	int64
uint64_t	C.uint64_t	uint64

表 2-2 `<stdint.h>`类型对比

前文说过，如果C语言的类型是由多个关键字组成，则无法通过虚拟的“C”包直接访问(比如C语言的`unsigned short`不能直接通过`C.unsigned short`访问)。但是，在`<stdint.h>`中通过使用C语言的`typedef`关键字将`unsigned short`重新定义为`uint16_t`这样一个单词的类型后，我们就可以通过`C.uint16_t`访问原来的`unsigned short`类型了。对于比较复杂的C语言类型，推荐使用`typedef`关键字提供一个规则的类型命名，这样更利于在CGO中访问。

go 字符串和切片

在CGO生成的`_cgo_export.h`头文件中还会为Go语言的字符串、切片、字典、接口和管道等特有的数据类型生成对应的C语言类型：

```
typedef struct { const char *p; GoInt n; } GoString;
typedef void *GoMap;
typedef void *GoChan;
typedef struct { void *t; void *v; } GoInterface;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;
```

不过需要注意的是，其中只有字符串和切片在CGO中有一定的使用价值，因为CGO为他们的某些GO语言版本的操作函数生成了C语言版本，因此二者可以在Go调用C语言函数时马上使用;而CGO并未针对其他的类型提供相关的辅助函数，且Go语言特有的内存模型导致我们无法保持这些由Go语言管理的内存指针，所以它们C语言环境并无使用的价值。

在导出的C语言函数中我们可以直接使用Go字符串和切片。假设有以下两个导出函数：

```
//export helloString
func helloString(s string) {}

//export helloSlice
func helloSlice(s []byte) {}
```

CGO生成的`_cgo_export.h`头文件会包含以下的函数声明：

```
extern void helloString(GoString p0);
extern void helloSlice(GoSlice p0);
```

不过需要注意的是，如果使用了GoString类型则会对`_cgo_export.h`头文件产生依赖，而这个头文件是动态输出的。

结构体、联合、枚举类型

C语言的结构体、联合、枚举类型不能作为匿名成员被嵌入到Go语言的结构体中。在Go语言中，我们可以通过`C.struct_xxx`来访问C语言中定义的`struct xxx`结构体类型。结构体的内存布局按照C语言的通用对齐规则，在32位Go语言环境C语言结构体也按照32位对齐规则，在64位Go语言环境按照64位的对齐规则。对于指定了特殊对齐规则的结构体，无法在CGO中访问。

结构体的简单用法如下：

```
package main

/*
struct A {
    int i;
    float f;
};
*/
import "C"
import "fmt"

func main() {
    var a C.struct_A
    a.i = 3
    a.f = 3.5
    fmt.Println(a.i)
    fmt.Println(a.f)
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
3
3.5
```

如果结构体的成员名字中碰巧是Go语言的关键字，可以通过在成员名开头添加下划线来访问：

```
/*
struct A {
    int type; // type 是 Go 语言的关键字
};
*/
import "C"
import "fmt"

func main() {
    var a C.struct_A
    a._type=10
    fmt.Println(a._type) // _type 对应 type
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
10
```

但是如果有2个成员：一个是以Go语言关键字命名，另一个刚好是以下划线和Go语言关键字命名，那么以Go语言关键字命名的成员将无法访问（被屏蔽）：

```
/*
struct A {
    int   type;  // type 是 Go 语言的关键字
    float _type; // 将屏蔽CGO对 type 成员的访问
};
*/
import "C"
import "fmt"

func main() {
    var a C.struct_A
    a._type = 3.5
    fmt.Println(a._type) // _type 对应 _type
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
3.5
```

C语言结构体中位字段对应的成员无法在Go语言中访问，如果需要操作位字段成员，需要通过在C语言中定义辅助函数来完成。对应零长数组的成员，无法在Go语言中直接访问数组的元素，但其中零长的数组成员所在位置的偏移量依然可以通过`unsafe.Offsetof(a.arr)`来访问。

```
/*
struct A {
    int   size: 10; // 位字段无法访问
    float arr[];    // 零长的数组也无法访问
};
*/
```

```
import "C"
import "fmt"

func main() {
    var a C.struct_A
    fmt.Println(a.size) // 错误: 位字段无法访问
    fmt.Println(a.arr)  // 错误: 零长的数组也无法访问
}
```

```
# command-line-arguments
.\easy.go:64:16: a.size undefined (type _Ctype_struct_A has no field or method size)
.\easy.go:65:16: a.arr undefined (type _Ctype_struct_A has no field or method arr)
```

在C语言中，我们无法直接访问Go语言定义的结构体类型。

对于联合类型，我们可以通过`C.union_xxx`来访问C语言中定义的`union xxx`类型。但是Go语言中并不支持C语言联合类型，它们会被转为对应大小的字节数组。

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

union B1 {
    int i;
    float f;
};

union B2 {
    int8_t i8;
    int64_t i64;
};
*/
import "C"
import "fmt"

func main() {
    var b1 C.union_B1;
    fmt.Printf("%T\n", b1) // [4]uint8

    var b2 C.union_B2;
    fmt.Printf("%T\n", b2) // [8]uint8
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
[4]uint8
[8]uint8
```

如果需要操作C语言的联合类型变量，一般有三种方法：第一种是在C语言中定义辅助函数；第二种是通过Go语言的"encoding/binary"手工解码成员(需要注意大端小端问题)；第三种是使用`unsafe`包强制转型为对应类型(这是性能最好的方式)。下面展示通过`unsafe`包访问联合类型成员的方式：

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

```
union B {
    int i;
    float f;
};
*/
import "C"
import "fmt"

func main() {
    var b C.union_B;
    fmt.Println("b.i:", *(*C.int)(unsafe.Pointer(&b)))
    fmt.Println("b.f:", *(*C.float)(unsafe.Pointer(&b)))
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
b.i: 0
b.f: 0
```

虽然`unsafe`包访问最简单、性能也最好，但是对于有嵌套联合类型的情况处理会导致问题复杂化。对于复杂的联合类型，推荐通过在C语言中定义辅助函数的方式处理。

对于枚举类型，我们可以通过`C.enum_xxx`来访问C语言中定义的`enum xxx`结构体类型。

```
/*
enum C {
    ONE,
    TWO,
};
*/
import "C"
import "fmt"

func main() {
    var c C.enum_C = C.TWO
    fmt.Println(c)
    fmt.Println(C.ONE)
    fmt.Println(C.TWO)
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_easy_go.exe
1
0
1
```

在C语言中，枚举类型底层对应`int`类型，支持负数类型的值。我们可以通过`C.ONE`、`C.TWO`等直接访问定义的枚举值。

数组、字符串和切片

在C语言中，数组名其实对应于一个指针，指向特定类型特定长度的一段内存，但是这个指针不能被修改；当把数组名传递给一个函数时，实际上传递的是数组第一个元素的地址。为了讨论方便，我们将一段特定长度的内存统称为数组。C语言的字符串是一个char类型的数组，字符串的长度需要根据表示结尾的NULL字符的位置确定。C语言中没有切片类型。

在Go语言中，数组是一种值类型，而且数组的长度是数组类型的一个部分。Go语言字符串对应一段长度确定的只读byte类型的内存。Go语言的切片则是一个简化版的动态数组。

Go语言和C语言的数组、字符串和切片之间的相互转换可以简化为Go语言的切片和C语言中指向一定长度内存的指针之间的转换。

CGO的C虚拟包提供了以下一组函数，用于Go语言和C语言之间数组和字符串的双向转换：

```
// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string

// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

其中`C.CString`针对输入的Go字符串，克隆一个C语言格式的字符串；返回的字符串由C语言的`malloc`函数分配，不使用时需要通过C语言的`free`函数释放。`C.CBytes`函数的功能和`C.CString`类似，用于从输入的Go语言字节切片克隆一个C语言版本的字节数组，同样返回的数组需要在合适的时候释放。`C.GoString`用于将从NULL结尾的C语言字符串克隆一个Go语言字符串。`C.GoStringN`是另一个字符数组克隆函数。`C.GoBytes`用于从C语言数组，克隆一个Go语言字节切片。

该组辅助函数都是以克隆的方式运行。当Go语言字符串和切片向C语言转换时，克隆的内存由C语言的`malloc`函数分配，最终可以通过`free`函数释放。当C语言字符串或数组向Go语言转换时，克隆的内存由Go语言分配管理。通过该组转换函数，转换前和转换后的内存依然在各自的语言环境中，它们并没有跨越Go语言和C语言。克隆方式实现转换的优点是接口和内存管理都很简单，缺点是克隆需要分配新的内存和复制操作都会导致额外的开销。

在`reflect`包中有字符串和切片的定义：

```
type StringHeader struct {
    Data uintptr
    Len  int
}

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

如果不希望单独分配内存，可以在Go语言中直接访问C语言的内存空间：

```
package main

/*
static char arr[10];
static char *s = "Hello";
*/
import "C"
import (
    "reflect"
    "unsafe"
)

func main() {
    // 通过 reflect.SliceHeader 转换
    var arr0 []byte
    var arr0Hdr = (*reflect.SliceHeader)(unsafe.Pointer(&arr0))
    arr0Hdr.Data = uintptr(unsafe.Pointer(&C.arr[0]))
    arr0Hdr.Len = 10
    arr0Hdr.Cap = 10

    // 通过切片语法转换
    arr1 := (*[31]byte)(unsafe.Pointer(&C.arr[0]))[:10:10]

    var s0 string
    var s0Hdr = (*reflect.StringHeader)(unsafe.Pointer(&s0))
    s0Hdr.Data = uintptr(unsafe.Pointer(C.s))
    s0Hdr.Len = int(C.strlen(C.s))

    sLen := int(C.strlen(C.s))
    s1 := string((*[31]byte)(unsafe.Pointer(&C.s[0]))[:sLen:sLen])

    println(arr1)
    println(s1)
}
```

感觉没有实现这个方法

```
# command-line-arguments
.\go_ack_c.go:27:18: could not determine kind of name for C.strlen
```

因为Go语言的字符串是只读的，用户需要自己保证Go字符串在使用期间，底层对应的C字符串内容不会发生变化、内存不会被提前释放掉。

在CGO中，会为字符串和切片生成和上面结构对应的C语言版本的结构体：

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

在C语言中可以通过`GoString`和`GoSlice`来访问Go语言的字符串和切片。如果是Go语言中数组类型，可以将数组转为切片后再行转换。如果字符串或切片对应的底层内存空间由Go语言的运行时管理，那么在C语言中不能长时间保存Go内存对象。

指针之间的转换

在C语言中，不同类型的指针是可以显式或隐式转换的，如果是隐式只是会在编译时给出一些警告信息。但是Go语言对于不同类型的转换非常严格，任何C语言中可能出现的警告信息在Go语言中都可能是错误！指针是C语言的灵魂，指针间的自由转换也是cgo代码中经常要解决的第一个重要的问题。

在Go语言中两个指针的类型完全一致则不需要转换可以直接通用。如果一个指针类型是用type命令在另一个指针类型基础之上构建的，换言之两个指针底层是相同完全结构的指针，那么我我们可以通过直接强制转换语法进行指针间的转换。但是cgo经常要面对的是2个完全不同类型的指针间的转换，原则上这种操作在纯Go语言代码是严格禁止的。

cgo存在的一个目的就是打破Go语言的禁止，恢复C语言应有的指针的自由转换和指针运算。以下代码演示了如何将X类型的指针转化为Y类型的指针：

```
var p *X
var q *Y

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

为了实现X类型指针到Y类型指针的转换，我们需要借助`unsafe.Pointer`作为中间桥接类型实现不同类型指针之间的转换。`unsafe.Pointer`指针类型类似C语言中的`void*`类型的指针。

下面是指针间的转换流程的示意图：

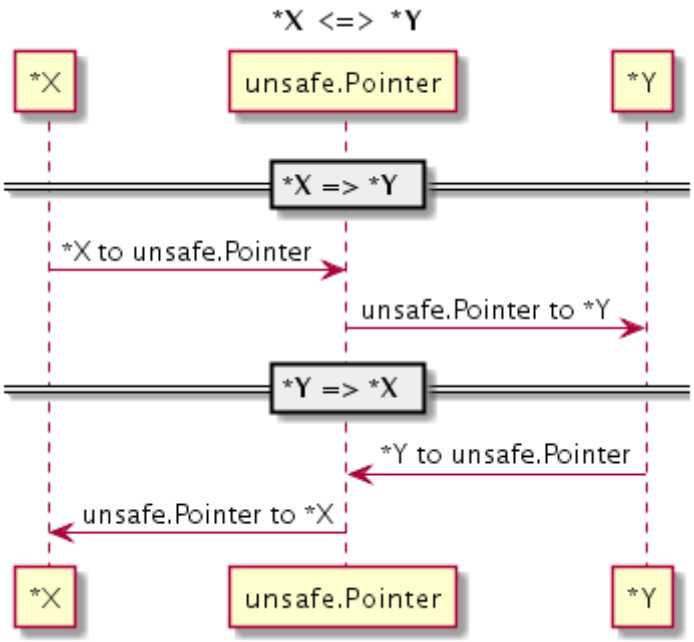


图 2-1 X类型指针转Y类型指针

任何类型的指针都可以通过强制转换为`unsafe.Pointer`指针类型去掉原有的类型信息，然后再重新赋予新的指针类型而达到指针间的转换的目的。

数值和指针的转换

不同类型指针间的转换看似复杂，但是在cgo中已经算是比较简单的了。在C语言中经常遇到用普通数值表示指针的场景，也就是说如何实现数值和指针的转换也是cgo需要面对的一个问题。

为了严格控制指针的使用，Go语言禁止将数值类型直接转为指针类型！不过，Go语言针对`unsafe.Pointr`指针类型特别定义了一个uintptr类型。我们可以uintptr为中介，实现数值类型到`unsafe.Pointr`指针类型到转换。再结合前面提到的方法，就可以实现数值和指针的转换了。

下面流程图演示了如何实现int32类型到C语言的`char*`字符串指针类型的相互转换：

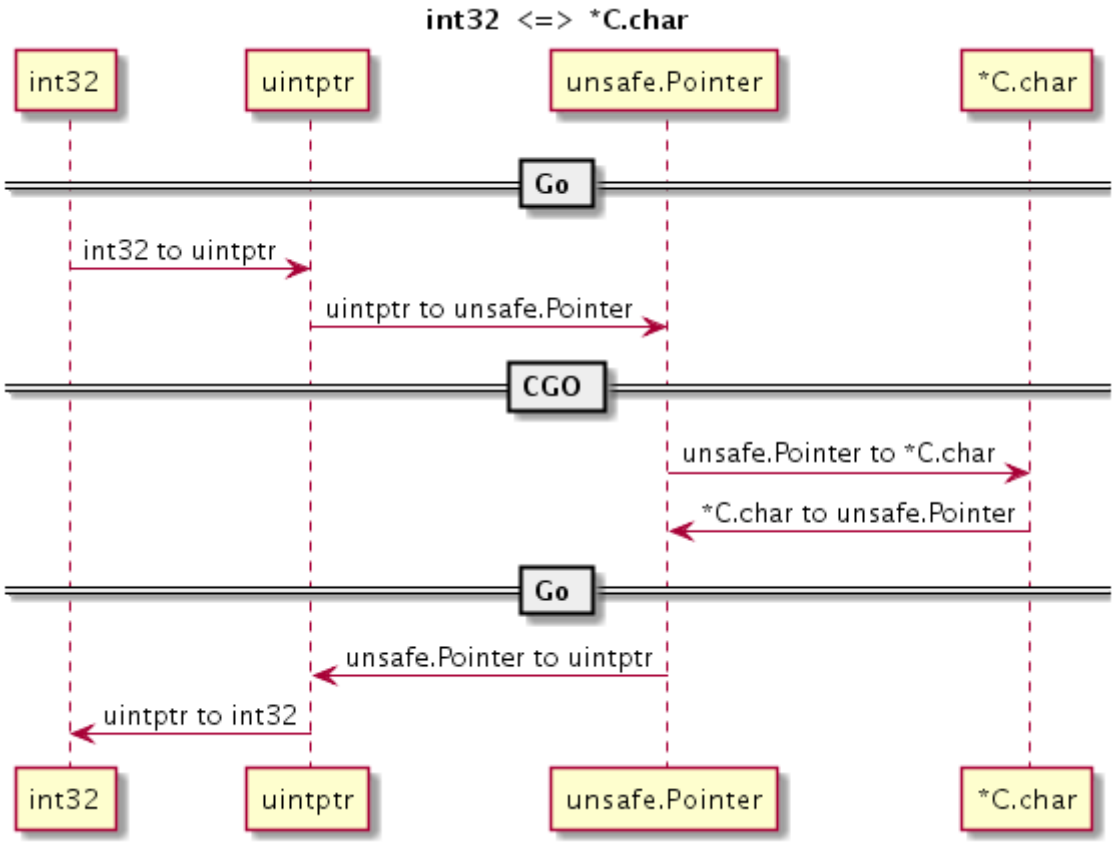


图 2-2 int32和`char*`指针转换

转换分为几个阶段，在每个阶段实现一个小目标：首先是int32到uintptr类型，然后是uintptr到`unsafe.Pointr`指针类型，最后是`unsafe.Pointr`指针类型到`*C.char`类型。

切片间的转换

在C语言中数组也一种指针，因此两个不同类型数组之间的转换和指针间转换基本类似。但是在Go语言中，数组或数组对应的切片都不再是指针类型，因此我们也就无法直接实现不同类型的切片之间的转换。

不过Go语言的reflect包提供了切片类型的底层结构，再结合前面讨论到不同类型之间的指针转换技术就可以实现`[]X`和`[]Y`类型的切片转换：

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

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])
```

不同切片类型之间转换的思路是先构造一个空的目标切片，然后用原有的切片底层数据填充目标切片。如果X和Y类型的大小不同，需要重新设置Len和Cap属性。需要注意的是，如果X或Y是空类型，上述代码中可能导致除0错误，实际代码需要根据情况酌情处理。

下面演示了切片间的转换的具体流程：

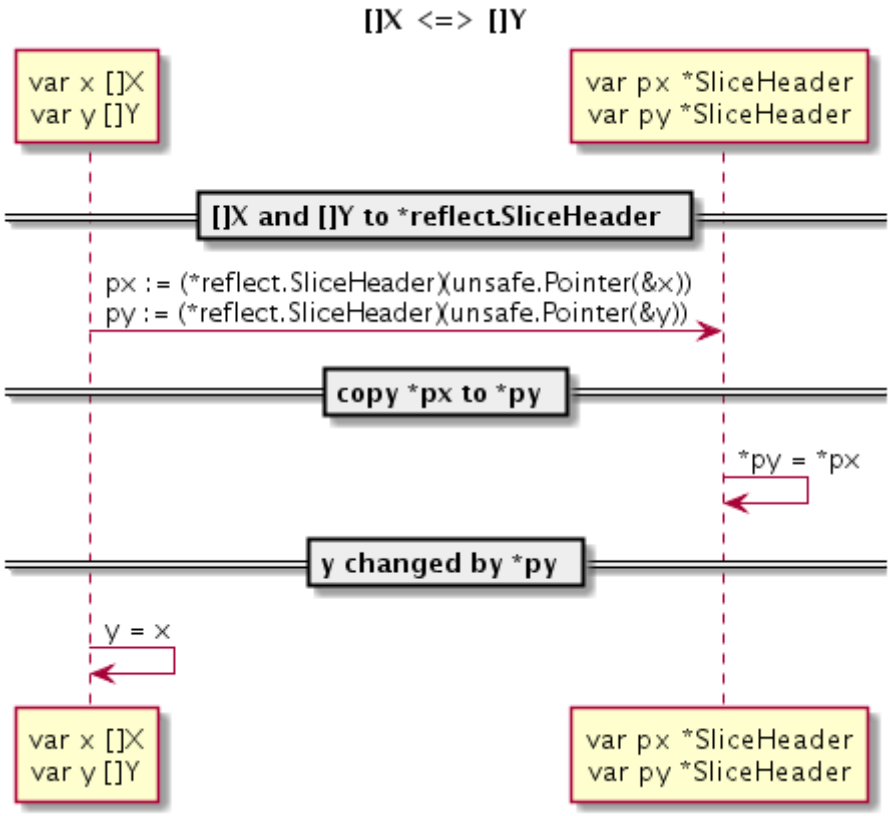


图 2-3 X类型切片转Y类型切片

函数调用

go调用C函数

对于一个启用CGO特性的程序，CGO会构造一个虚拟的C包。通过这个虚拟的C包可以调用C语言函数。

```
package main

import "C"

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

```
*/
import "C"

func main() {
   println(C.add(3, 5))
}
```

以上的CGO代码首先定义了一个当前文件内可见的add函数，然后通过`C.add`

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_go_add_go.exe
8
```

c函数的返回值

对于有返回值的C函数，我们可以正常获取返回值。

```
package main

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

func main() {
   println(C.div(6, 3))
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_go_div_go.exe
2
```

上面的div函数实现了一个整数除法的运算，然后通过返回值返回除法的结果。

不过对于除数为0的情形并没有做特殊处理。如果希望在除数为0的时候返回一个错误，其他时候返回正常的结果。因为C语言不支持返回多个结果，因此`<errno.h>`标准库提供了一个`errno`宏用于返回错误状态。我们可以近似地将`errno`看成一个线程安全的全局变量，可以用于记录最近一次错误的状态码。

改进后的div函数实现如下：

```
#include <errno.h>

int div(int a, int b) {
    if(b == 0) {
        errno = EINVAL;
        return 0;
    }
    return a/b;
}
```

CGO也针对`<errno.h>`标准库的`errno`宏做的特殊支持：在CGO调用C函数时如果有两个返回值，那么第二个返回值将对应`errno`错误状态。

```
package main

/*
#include <errno.h>

static int div(int a,int b){
if(b==0){
   errno = EINVAL;
   return 0;
}
   return a/b;
}
*/
import "C"
import "fmt"

func main() {
   v0, err0 := C.div(2, 1)
   fmt.Println(v0, err0)

   v1, err1 := C.div(1, 0)
   fmt.Println(v1, err1)
}
```

运行这个代码将会产生以下输出：

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_go_div_modify_go.exe
2 <nil>
0 The device does not recognize the command.
```

我们可以近似地将div函数看作为以下类型的函数：

```
func C.div(a, b C.int) (C.int, [error])
```

第二个返回值是可忽略的error接口类型，底层对应 `syscall.Errno` 错误类型。

void函数的返回值

C语言函数还有一种没有返回值类型的函数，用void表示返回值类型。一般情况下，我们无法获取void类型函数的返回值，因为没有返回值可以获取。前面的例子中提到，cgo对errno做了特殊处理，可以通过第二个返回值来获取C语言的错误状态。对于void类型函数，这个特性依然有效。

以下的代码是获取没有返回值函数的错误状态码：C语言函数还有一种没有返回值类型的函数，用void表示返回值类型。一般情况下，我们无法获取void类型函数的返回值，因为没有返回值可以获取。前面的例子中提到，cgo对errno做了特殊处理，可以通过第二个返回值来获取C语言的错误状态。对于void类型函数，这个特性依然有效。

以下的代码是获取没有返回值函数的错误状态码：

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

func main() {
    _, err := C.noreturn()
    fmt.Println(err)
}
```

此时，我们忽略了第一个返回值，只获取第二个返回值对应的错误码。

我们也可以尝试获取第一个返回值，它对应的是C语言的void对应的Go语言类型：

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

func main() {
    v, _ := C.noreturn()
    fmt.Printf("%#v", v)
}
```

运行这个代码将会产生以下输出：

```
main._Ctype_void{}
```

我们可以看出C语言的void类型对应的是当前的main包中的`_Ctype_void`类型。其实也将C语言的noreturn函数看作是返回`_Ctype_void`类型的函数，这样就可以直接获取void类型函数的返回值：

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

func main() {
    fmt.Println(C.noreturn())
}
```

运行这个代码将会产生以下输出：

```
[]
```

其实在CGO生成的代码中，`_Ctype_void`类型对应一个0长的数组类型`[0]byte`，因此`fmt.Println`输出的是一个表示空数值的方括弧。

c调用go导出函数

CGO还有一个强大的特性：将Go函数导出为C语言函数。这样的话我们可以定义好C语言接口，然后通过Go语言实现。在本章的第一节快速入门部分我们已经展示过Go语言导出C语言函数的例子。

下面是用Go语言重新实现本节开始的add函数：

```
import "C"

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

add函数名以小写字母开头，对于Go语言来说是包内的私有函数。但是从C语言角度来看，导出的add函数是一个可全局访问的C语言函数。如果在两个不同的Go语言包内，都存在一个同名的要导出为C语言函数的add函数，那么在最终的链接阶段将会出现符号重名的问题。

CGO生成的 `_cgo_export.h` 文件回包含导出后的C语言函数的声明。我们可以在纯C源文件中包含 `_cgo_export.h` 文件来引用导出的add函数。如果希望在当前的CGO文件中马上使用导出的C语言add函数，则无法引用 `_cgo_export.h` 文件。因为 `_cgo_export.h` 文件的生成需要依赖当前文件可以正常构建，而如果当前文件内部循环依赖还未生成的 `_cgo_export.h` 文件将会导致cgo命令错误。

```
#include "_cgo_export.h"

void foo() {
    add(1, 1);
}
```

当导出C语言接口时，需要保证函数的参数和返回值类型都是C语言友好的类型，同时返回值不得直接或间接包含Go语言内存空间的指针。

内部机制

CGO生成的中间件

要了解CGO技术的底层秘密首先需要了解CGO生成了哪些中间文件。我们可以在构建一个cgo包时增加一个 `-work` 输出中间生成文件所在的目录并且在构建完成时保留中间文件。如果是比较简单的cgo代码我们也可以直接通过手工调用 `go tool cgo` 命令来查看生成的中间文件。

在一个Go源文件中，如果出现了 `import "C"` 指令则表示将调用cgo命令生成对应的中间文件。下图是cgo生成的中间文件的简单示意图：

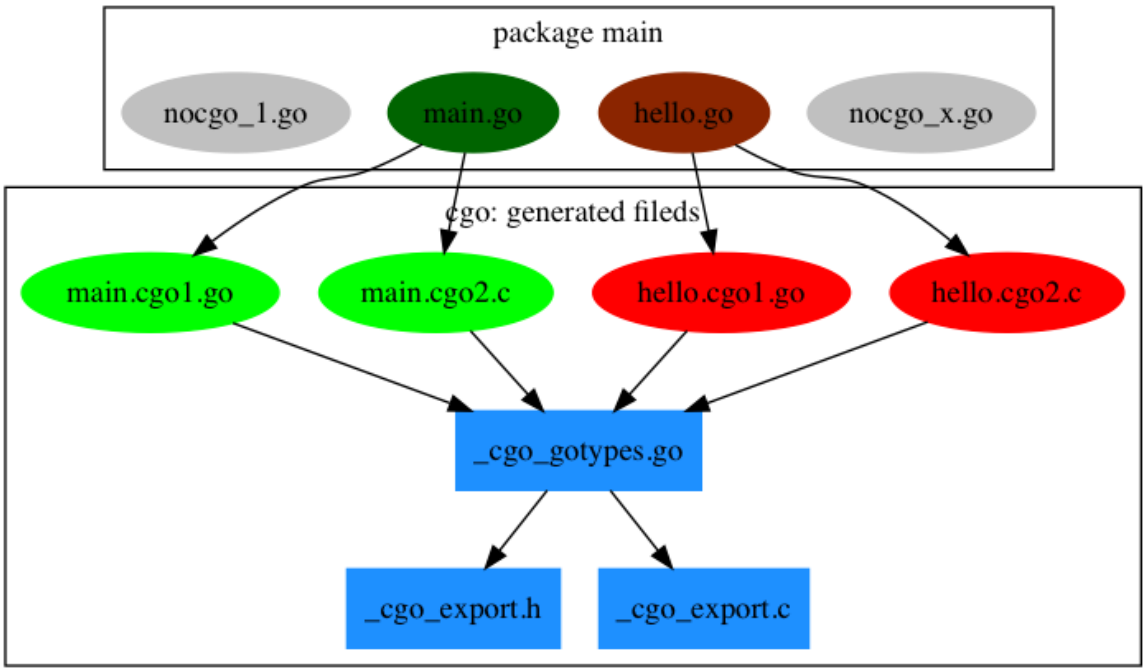


图 cgo生成的中间文件

包中有4个Go文件，其中nocgo开头的文件中没有 `import "C"` 指令，其它的2个文件则包含了cgo代码。cgo命令会为每个包含了cgo代码的Go文件创建2个中间文件，比如 main.go 会分别创建 main.cgo1.go 和 main.cgo2.c 两个中间文件。然后会为整个包创建一个 `_cgo_gotypes.go` Go文件，其中包含Go语言部分辅助代码。此外还会创建一个 `_cgo_export.h` 和 `_cgo_export.c` 文件，对应Go语言导出到C语言的类型和函数。

GO调用C函数

```
package main

//int sum(int a,int b){return a+b;}
import "C"

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

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_go_use_c_go.exe
4
```

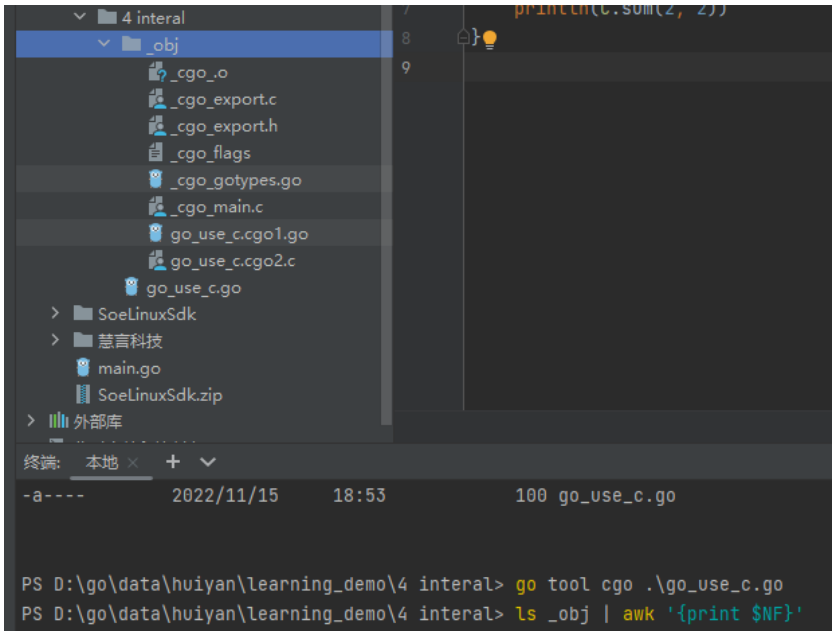
首先构建并运行该例子没有错误。然后通过cgo命令行工具在_obj目录生成中间文件：

```
$ go tool cgo main.go
```

查看_obj目录生成中间文件：

```
$ ls _obj | awk '{print $NF}'
_cgo_.o
_cgo_export.c
_cgo_export.h
_cgo_flags
_cgo_gotypes.go
_cgo_main.c
main.cgo1.go
main.cgo2.c
```

其中`_cgo_.o`、`_cgo_flags`和`_cgo_main.c`文件和我们的代码没有直接的逻辑关联，可以暂时忽略。



我们先查看`main.cgo1.go`文件，它是main.go文件展开虚拟C包相关函数和变量后的Go代码：

```
package main

//int sum(int a, int b) { return a+b; }
import _ "unsafe"

func main() {
    println((_Cfunc_sum)(1, 1))
}
```

其中`C.sum(1, 1)`函数调用被替换成了`(_Cfunc_sum)(1, 1)`。每一个`C.xxx`形式的函数都会被替换为`_Cfunc_xxx`格式的纯Go函数，其中前缀`_Cfunc_`表示这是一个C函数，对应一个私有的Go桥接函数。

`_Cfunc_sum`函数在cgo生成的`_cgo_gotypes.go`文件中定义：

```
//go:cgo_unsafe_args
func _Cfunc_sum(p0 _Ctype_int, p1 _Ctype_int) (r1 _Ctype_int) {
    _cgo_runtime_cgocall(_cgo_506f45f9fa85_Cfunc_sum,
uintptr(unsafe.Pointer(&p0)))
    if _Cgo_always_false {
        _Cgo_use(p0)
        _Cgo_use(p1)
    }
    return
}
```

`_Cfunc_sum`函数的参数和返回值`_Ctype_int`类型对应`C.int`类型，命名的规则和`_Cfunc_xxx`类似，不同的前缀用于区分函数和类型。

其中`_cgo_runtime_cgocall`对应`runtime.cgocall`函数，函数的声明如下：

```
func runtime.cgocall(fn, arg unsafe.Pointer) int32
```

第一个参数是C语言函数的地址，第二个参数是存放C语言函数对应的参数结构体的地址。

在这个例子中，被传入C语言函数`_cgo_506f45f9fa85_Cfunc_sum`也是cgo生成的中间函数。函数在`main.cgo2.c`定义：

```
void _cgo_506f45f9fa85_Cfunc_sum(void *v) {
    struct {
        int p0;
        int p1;
        int r;
        char __pad12[4];
    } __attribute__((__packed__)) *a = v;
    char *stktop = _cgo_topofstack();
    __typeof__(a->r) r;
    _cgo_tsan_acquire();
    r = sum(a->p0, a->p1);
    _cgo_tsan_release();
    a = (void*)((char*)a + (_cgo_topofstack() - stktop));
    a->r = r;
}
```

这个函数参数只有一个void范型的指针，函数没有返回值。真实的sum函数的函数参数和返回值均通过唯一的参数指针类实现。

`_cgo_506f45f9fa85_Cfunc_sum`函数的指针指向的结构为：

```
    struct {
        int p0;
        int p1;
        int r;
        char __pad12[4];
    } __attribute__((__packed__)) *a = v;
```

其中p0成员对应sum的第一个参数，p1成员对应sum的第二个参数，r成员，`__pad12`用于填充结构体保证对齐CPU机器字的整倍数。

然后从参数指向的结构体获取调用参数后开始调用真实的C语言版sum函数，并且将返回值保持到结构体内返回值对应的成员。

因为Go语言和C语言有着不同的内存模型和函数调用规范。其中`_cgo_topofstack`函数相关的代码用于C函数调用后恢复调用栈。`_cgo_tsan_acquire`和`_cgo_tsan_release`则是用于扫描CGO相关的函数则是对CGO相关函数的指针做相关检查。

`C.sum`的整个调用流程图如下：

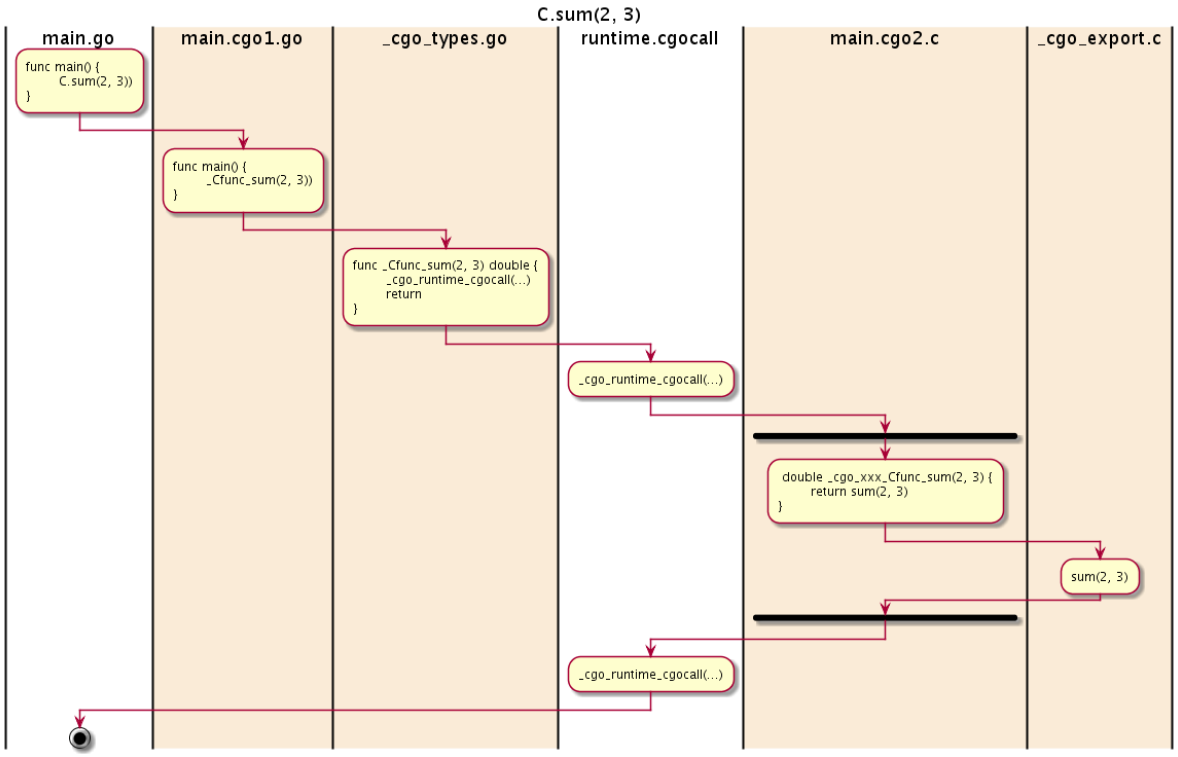


图 2-5 调用C函数

其中`runtime.cgocall`函数是实现Go语言到C语言函数跨界调用的关键。更详细的细节可以参考 https://golang.org/src/cmd/cgo/doc.go 内部的代码注释和 `runtime.cgocall` 函数的实现。

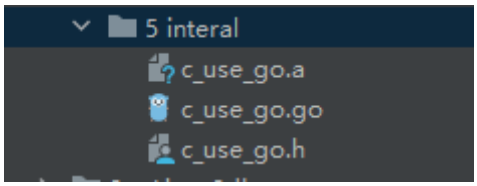
C调用GO函数

在简单分析了Go调用C函数的流程后，我们现在来分析C反向调用Go函数的流程。同样，我们现构造一个Go语言版本的sum函数，文件名同样为`main.go`：

```
package main

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

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

func main() {
   println(sum(32, 3))
}
```

```
C:\Users\Administrator\AppData\Local\Temp\GoLand\___go_build_c_use_go_go.exe
35
```

CGO的语法细节不在赘述。为了在C语言中使用sum函数，我们需要将Go代码编译为一个C静态库：



要分析生成的C语言版sum函数的调用流程，同样需要分析cgo生成的中间文件：

```
$ go tool cgo c_use_go.go
```

_obj目录还是生成类似的中间文件。为了查看方便，我们刻意忽略了无关的几个文件：

```
$ ls _obj | awk '{print $NF}'
_cgo_export.c
_cgo_export.h
_cgo_gotypes.go
main.cgo1.go
main.cgo2.c
```

其中`_cgo_export.h`文件的内容和生成C静态库时产生的`sum.h`头文件是同一个文件，里面同样包含sum函数的声明。

既然C语言是主调用者，我们需要先从C语言版sum函数的实现开始分析。C语言版本的sum函数在生成的`_cgo_export.c`文件中（该文件包含的是Go语言导出函数对应的C语言函数实现）：

```
int sum(int p0, int p1)
{
    __SIZE_TYPE__ _cgo_ctxt = _cgo_wait_runtime_init_done();
    struct {
        int p0;
        int p1;
        int r0;
        char __pad0[4];
    } __attribute__((__packed__)) a;
    a.p0 = p0;
    a.p1 = p1;
    _cgo_tsan_release();
    crosscall2(_cgoexp_8313eaf44386_sum, &a, 16, _cgo_ctxt);
    _cgo_tsan_acquire();
    _cgo_release_context(_cgo_ctxt);
    return a.r0;
}
```

sum函数的内容采用和前面类似的技术，将sum函数的参数和返回值打包到一个结构体中，然后通过`runtime/cgo.crosscall2`函数将结构体传给`_cgoexp_8313eaf44386_sum`函数执行。

`runtime/cgo.crosscall2`函数采用汇编语言实现，它对应的函数声明如下：

```
func runtime/cgo.crosscall2(
    fn func(a unsafe.Pointer, n int32, ctxt uintptr),
    a unsafe.Pointer, n int32,
    ctxt uintptr,
)
```

其中关键的是fn和a，fn是中间代理函数的指针，a是对应调用参数和返回值的结构体指针。

中间的`_cgoexp_8313eaf44386_sum`代理函数在`_cgo_gotypes.go`文件：

```
func _cgoexp_8313eaf44386_sum(a unsafe.Pointer, n int32, ctxt uintptr) {
    fn := _cgoexpwrap_8313eaf44386_sum
    _cgo_runtime_cgocallback(**(**unsafe.Pointer)(unsafe.Pointer(&fn)), a,
uintptr(n), ctxt);
}

func _cgoexpwrap_8313eaf44386_sum(p0 _Ctype_int, p1 _Ctype_int) (r0 _Ctype_int)
{
    return sum(p0, p1)
}
```

内部将sum的包装函数`_cgoexpwrap_8313eaf44386_sum`作为函数指针，然后由`_cgo_runtime_cgocallback`函数完成C语言到Go函数的回调工作。

`_cgo_runtime_cgocallback`函数对应`runtime.cgocallback`函数，函数的类型如下：

```
func runtime.cgocallback(fn, frame unsafe.Pointer, framesize, ctxt uintptr)
```

参数分别是函数指针，函数参数和返回值对应结构体的指针，函数调用帧大小和上下文参数。

整个调用流程图如下：

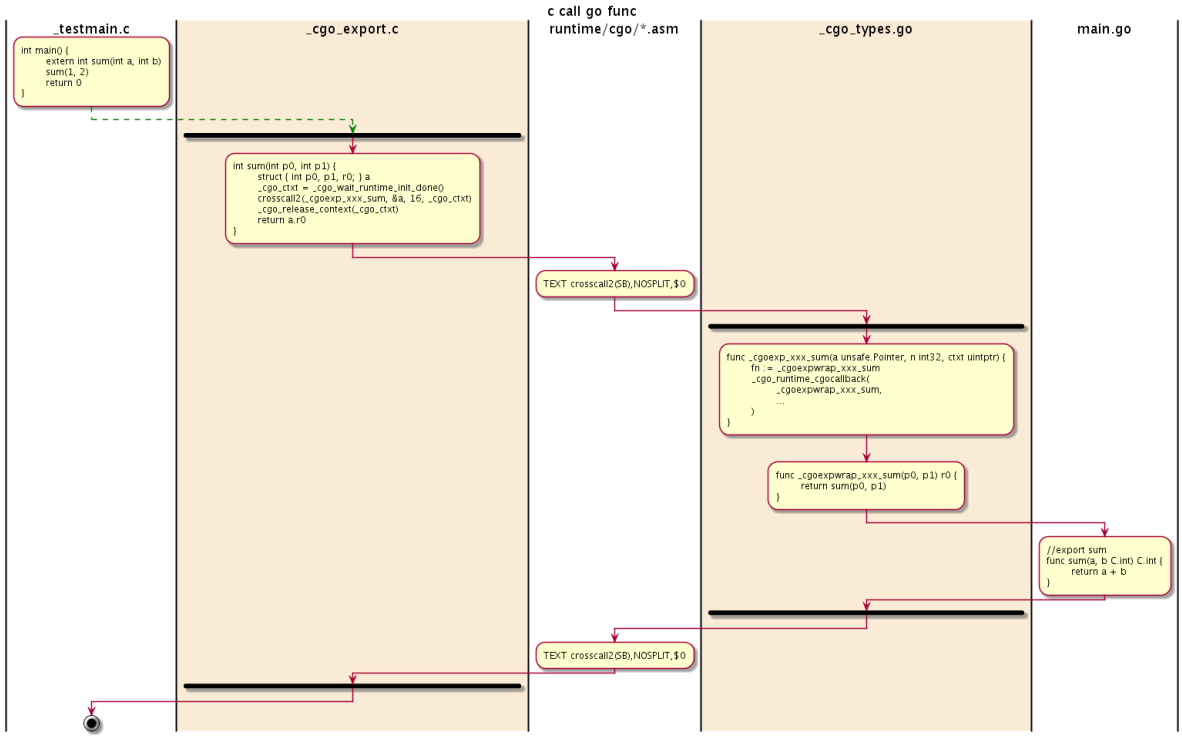


图 2-6 调用导出的Go函数

其中`runtime.cgocallback`函数是实现C语言到Go语言函数跨界调用的关键。更详细的细节可以参考相关函数的实现。

任务布置的理解

用户传数据 失败之后返回信息

go build -o 生成动态链接库

动态链接库的方法在main.go里面定义

4个回调方法，有些函数不需要回调

构建的流程：

传参是C的数据类型 ->转换成go的数据类型->进行逻辑操作->打包成dll

任务布置：写一个C字符串进来 转成go 然后printf

> 就是现在的项目不是有个c的demo，调用go导出的dll，你也可以照着他的写，先export个方法，可以传进来个字符串，你打印下，或者传个数组进来你排序下，然后生成为dll，在c里调用测试下可以

参考资料

https://books.studygolang.com/advanced-go-programming-book/ch2-cgo/ch2-01-hello-cgo.html