C/C++经过几十年的发展，已经积累了庞大的软件资产，它们很多久经考验而且性能已经足够优化。Go语言必须能够站在C/C++这个巨人的肩膀之上，有了海量的C/C++软件资产兜底之后，我们才可以放心愉快地用Go语言编程。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。Go语言通过自带的一个叫CGO的工具来支持C语言函数调用，同时我们可以用Go语言导出C动态库接口给其它语言使用。本章主要讨论CGO编程中涉及的一些问题。

# 1. 快速入门

本节我们将通过一系列由浅入深的小例子来快速掌握CGO的基本用法。

## 1.1 最简CGO程序

真实的CGO程序一般都比较复杂。不过我们可以由浅入深，一个最简的CGO程序该是什么样的呢？要构造一个最简CGO程序，首先要忽视一些复杂的CGO特性，同时要展示CGO程序和纯Go程序的差别来。下面是我们构建的最简CGO程序：

// hello.go  
package main  
  
import "C"  
  
func main() {  
 println("hello cgo")  
}

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。

## 1.2 基于C标准库函数输出字符串

第一章那个CGO程序还不够简单，我们现在来看看更简单的版本：

// hello.go  
package main  
  
//#include <stdio.h>  
import "C"  
  
func main() {  
 C.puts(C.CString("Hello, World\n"))  
}

我们不仅仅通过import "C"语句启用CGO特性，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。

相比“Hello, World 的革命”一节中的CGO程序最大的不同是：我们没有在程序退出前释放C.CString创建的C语言字符串；还有我们改用puts函数直接向标准输出打印，之前是采用fputs向标准输出打印。

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。

## 1.3 使用自己的C函数

前面我们使用了标准库中已有的函数。现在我们先自定义一个叫SayHello的C函数来实现打印，然后从Go语言环境中调用这个SayHello函数：

除了SayHello函数是我们自己实现的之外，其它的部分和前面的例子基本相似。

我们也可以将SayHello函数放到当前目录下的一个C语言源文件中（后缀名必须是.c）。因为是编写在独立的C文件中，为了允许外部引用，所以需要去掉函数的static修饰符。

// hello.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"))  
}

然后在CGO部分先声明SayHello函数，其它部分不变：

// hello.go  
package main  
  
//void SayHello(const char\* s);  
import "C"  
  
func main() {  
 C.SayHello(C.CString("Hello, World\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。

既然SayHello函数已经放到独立的C文件中了，我们自然可以将对应的C文件编译打包为静态库或动态库文件供使用。如果是以静态库或动态库方式引用SayHello函数的话，需要将对应的C源文件移出当前目录（CGO构建程序会自动构建当前目录下的C源文件，从而导致C函数名冲突）。关于静态库等细节将在稍后章节讲解。

## 1.4 C代码的模块化

在编程过程中，抽象和模块化是将复杂问题简化的通用手段。当代码语句变多时，我们可以将相似的代码封装到一个个函数中；当程序中的函数变多时，我们将函数拆分到不同的文件或模块中。而模块化编程的核心是面向程序接口编程（这里的接口并不是Go语言的interface，而是API的概念）。

在前面的例子中，我们可以抽象一个名为hello的模块，模块的全部接口函数都在hello.h头文件定义：

// hello.h  
void SayHello(const char\* s);

其中只有一个SayHello函数的声明。但是作为hello模块的用户来说，就可以放心地使用SayHello函数，而无需关心函数的具体实现。而作为SayHello函数的实现者来说，函数的实现只要满足头文件中函数的声明的规范即可。下面是SayHello函数的C语言实现，对应hello.c文件：

// hello.c  
  
#include "hello.h"  
#include <stdio.h>  
  
void SayHello(const char\* s) {  
 puts(s);  
}

在hello.c文件的开头，实现者通过#include "hello.h"语句包含SayHello函数的声明，这样可以保证函数的实现满足模块对外公开的接口。

接口文件hello.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：

// hello.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hello.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。

在采用面向C语言API接口编程之后，我们彻底解放了模块实现者的语言枷锁：实现者可以用任何编程语言实现模块，只要最终满足公开的API约定即可。我们可以用C语言实现SayHello函数，也可以使用更复杂的C++语言来实现SayHello函数，当然我们也可以用汇编语言甚至Go语言来重新实现SayHello函数。

## 1.5 用Go重新实现C函数

其实CGO不仅仅用于Go语言中调用C语言函数，还可以用于导出Go语言函数给C语言函数调用。在前面的例子中，我们已经抽象一个名为hello的模块，模块的全部接口函数都在hello.h头文件定义：

// hello.h  
void SayHello(/\*const\*/ char\* s);

现在我们创建一个hello.go文件，用Go语言重新实现C语言接口的SayHello函数:

// hello.go  
package main  
  
import "C"  
  
import "fmt"  
  
//***export*** SayHello  
func SayHello(s \*C.char) {  
 fmt.Print(C.GoString(s))  
}

我们**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，我们禁止了在函数的声明语句中的const修饰符。需要注意的是，这里其实有两个版本的SayHello函数：一个Go语言环境的；另一个是C语言环境的。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。

通过面向C语言接口的编程技术，我们不仅仅解放了函数的实现者，同时也简化的函数的使用者。现在我们可以将SayHello当作一个标准库的函数使用（和puts函数的使用方式类似）：

package main  
  
//#include <hello.h>  
import "C"  
  
func main() {  
 C.SayHello(C.CString("Hello, World\n"))  
}

一切似乎都回到了开始的CGO代码，但是代码内涵更丰富了。

## 1.6 面向C接口的Go编程

在开始的例子中，我们的全部CGO代码都在一个Go文件中。然后，通过面向C接口编程的技术将SayHello分别拆分到不同的C文件，而main依然是Go文件。再然后，是用Go函数重新实现了C语言接口的SayHello函数。但是对于目前的例子来说只有一个函数，要拆分到三个不同的文件确实有些繁琐了。

正所谓合久必分、分久必合，我们现在尝试将例子中的几个文件重新合并到一个Go文件。下面是合并后的成果：

package main  
  
//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))  
}

现在版本的CGO代码中C语言代码的比例已经很少了，但是我们依然可以进一步以Go语言的思维来提炼我们的CGO代码。通过分析可以发现SayHello函数的参数如果可以直接使用Go字符串是最直接的。在Go1.10中CGO新增加了一个\_GoString\_预定义的C语言类型，用来表示Go语言字符串。下面是改进后的代码：

// ***+build*** go1.10  
  
package main  
  
//void SayHello(\_GoString\_ s);  
import "C"  
  
import (  
 "fmt"  
)  
  
func main() {  
 C.SayHello("Hello, World\n")  
}  
  
//***export*** SayHello  
func SayHello(s string) {  
 fmt.Print(s)  
}

虽然看起来全部是Go语言代码，但是执行的时候是先从Go语言的main函数，到CGO自动生成的C语言版本SayHello桥接函数，最后又回到了Go语言环境的SayHello函数。这个代码包含了CGO编程的精华，读者需要深入理解。

# 2. CGO基础

要使用CGO特性，需要安装C/C++构建工具链，在macOS和Linux下是要安装GCC，在windows下是需要安装MinGW工具。同时需要保证环境变量CGO\_ENABLED被设置为1，这表示CGO是被启用的状态。在本地构建时CGO\_ENABLED默认是启用的，当交叉构建时CGO默认是禁止的。比如要交叉构建ARM环境运行的Go程序，需要手工设置好C/C++交叉构建的工具链，同时开启CGO\_ENABLED环境变量。然后通过import "C"语句启用CGO特性。

## 2.1. import “C”语句

如果在Go代码中出现了import "C"语句则表示使用了CGO特性，紧跟在这行语句前面的注释是一种特殊语法，里面包含的是正常的C语言代码。当确保CGO启用的情况下，还可以在当前目录中包含C/C++对应的源文件。

举个最简单的例子：

package main  
  
/\*  
#include <stdio.h>  
  
void printint(int v) {  
 printf("printint: %d\n", v);  
}  
\*/  
import "C"  
  
func main() {  
 v := 42  
 C.printint(C.int(v))  
}

这个例子展示了cgo的基本使用方法。开头的注释中写了要调用的C函数和相关的头文件，头文件被include之后里面的所有的C语言元素都会被加入到”C”这个虚拟的包中。需要注意的是，**import "C"导入语句需要单独一行，不能与其他包一同import。**向C函数传递参数也很简单，就直接转化成对应C语言类型传递就可以。如上例中C.int(v)用于将一个Go中的int类型值强制类型转换转化为C语言中的int类型值，然后调用C语言定义的printint函数进行打印。

需要注意的是，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。

比如我们希望在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。

有Go语言使用经验的用户可能会建议参数转型后再传入。但是这个方法似乎也是不可行的，因为cgo\_helper.PrintCString的参数是它自身包引入的\*C.char类型，在外部是无法直接获取这个类型的。换言之，一个包如果在公开的接口中直接使用了\*C.char等类似的虚拟C包的类型，其它的Go包是无法直接使用这些类型的，除非这个Go包同时也提供了\*C.char类型的构造函数。因为这些诸多因素，如果想在go test环境直接测试这些cgo导出的类型也会有相同的限制。

## 2.2.#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)。

对于在cgo环境混合使用C和C++的用户来说，可能有三种不同的编译选项：其中CFLAGS对应C语言特有的编译选项、CXXFLAGS对应是C++特有的编译选项、CPPFLAGS则对应C和C++共有的编译选项。但是在链接阶段，C和C++的链接选项是通用的，因此这个时候已经不再有C和C++语言的区别，它们的目标文件的类型是相同的。

#cgo指令还支持条件选择，当满足某个操作系统或某个CPU架构类型时后面的编译或链接选项生效。比如下面是分别针对windows和非windows下平台的编译和链接选项：

// #cgo windows CFLAGS: -DX86=1  
// #cgo !windows LDFLAGS: -lm

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))  
}

这样我们就可以用C语言中常用的技术来处理不同平台之间的差异代码。

## 2.3. 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。

比如下面的源文件只有在设置debug构建标志时才会被构建：

// ***+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。

# 3. 类型转换

最初CGO是为了达到方便从Go语言函数调用C语言函数（用C语言实现Go语言声明的函数）以复用C语言资源这一目的而出现的（因为C语言还会涉及回调函数，自然也会涉及到从C语言函数调用Go语言函数（用Go语言实现C语言声明的函数））。现在，它已经演变为C语言和Go语言双向通讯的桥梁。要想利用好CGO特性，自然需要了解此二语言类型之间的转换规则，这是本节要讨论的问题。

## 3.1. 数值类型

在Go语言中访问C语言的符号时，一般是通过虚拟的“C”包访问，比如C.int对应C语言的int类型。有些C语言的类型是由多个关键字组成，但通过虚拟的“C”包访问C语言类型时名称部分不能有空格字符，比如unsigned int不能直接通过C.unsigned int访问。因此CGO为C语言的基础数值类型都提供了相应转换规则，比如C.uint对应C语言的unsigned int。

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**。

## 3.2. 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。

Go1.10针对Go字符串增加了一个\_GoString\_预定义类型，可以降低在cgo代码中可能对\_cgo\_export.h头文件产生的循环依赖的风险。我们可以调整helloString函数的C语言声明为：

extern void helloString(\_GoString\_ p0);

因为\_GoString\_是预定义类型，我们无法通过此类型直接访问字符串的长度和指针等信息。Go1.10同时也增加了以下两个函数用于获取字符串结构中的长度和指针信息：

size\_t \_GoStringLen(\_GoString\_ s);  
const char \*\_GoStringPtr(\_GoString\_ s);

更严谨的做法是为C语言函数接口定义严格的头文件，然后基于稳定的头文件实现代码。

## 3.3. 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  
 fmt.Println(a.i)  
 fmt.Println(a.f)  
}

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：

package main  
  
/\*  
struct A {  
 int type; // type 是 Go 语言的关键字  
};  
\*/  
import "C"  
import "fmt"  
  
func main() {  
 var a C.struct\_A  
 fmt.Println(a.\_type) // \_type 对应 type  
}

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）：

package main  
  
/\*  
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  
 fmt.Println(a.\_type) // \_type 对应 \_type  
}

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。

package main  
  
/\*  
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  
}

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。

package main  
  
/\*  
#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  
}

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：

package main  
  
/\*  
#include <stdint.h>  
  
union B {  
 int i;  
 float f;  
};  
\*/  
import "C"  
import (  
 "fmt"  
 "unsafe"  
)  
  
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)))  
}

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。

package main  
  
/\*  
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)  
}

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。

## 3.4. 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：

package main  
  
// 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：

package main  
  
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：

/\*  
static char arr[10];  
static char \*s = "Hello";  
\*/  
import "C"  
import (  
 "fmt"  
 "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])  
}

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。

关于CGO内存模型的细节在稍后章节中会详细讨论。

## 3.5. 指针间的转换

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。

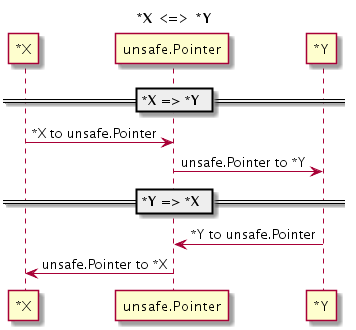
在Go语言中两个指针的类型完全一致则不需要转换可以直接通用。如果一个指针类型是用type命令在另一个指针类型基础之上构建的，换言之两个指针底层是相同完全结构的指针，那么我们可以通过直接强制转换语法进行指针间的转换。但是cgo经常要面对的是2个完全不同类型的指针间的转换，原则上这种操作在纯Go语言代码是严格禁止的。

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*

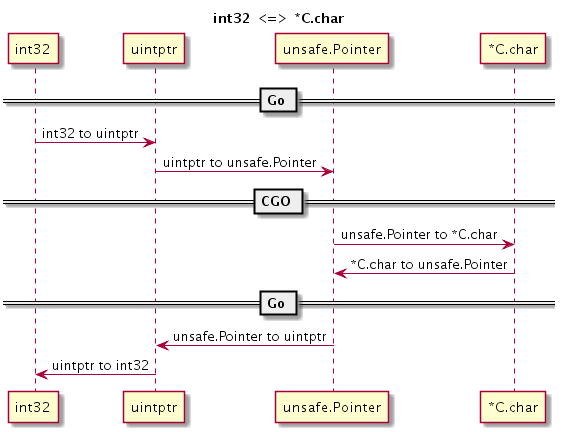
任何类型的指针都可以通过强制转换为unsafe.Pointer指针类型去掉原有的类型信息，然后再重新赋予新的指针类型而达到指针间的转换的目的。

## 3.6. 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。

## 3.7. 切片间的转换

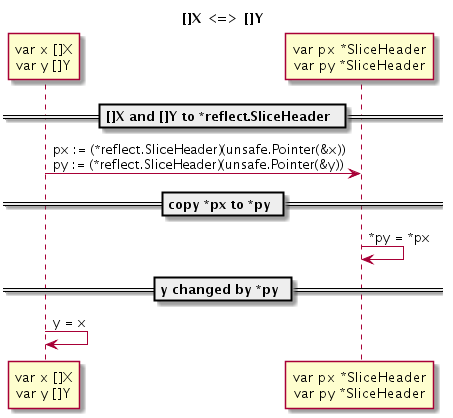
在C语言中数组也一种指针，因此两个不同类型数组之间的转换和指针间转换基本类似。但是在Go语言中，数组或数组对应的切片都不再是指针类型，因此我们也就无法直接实现不同类型的切片之间的转换。

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*

针对CGO中常用的功能，作者封装了 "github.com/chai2010/cgo" 包，提供基本的转换功能，具体的细节可以参考实现代码。

# 4. 函数调用

函数是C语言编程的核心，通过CGO技术我们不仅仅可以在Go语言中调用C语言函数，也可以将Go语言函数导出为C语言函数。

## 4.1. 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。

/\*  
static int add(int a, int b) {  
 return a+b;  
}  
\*/  
import "C"  
  
func main() {  
 C.add(1, 1)  
}

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。

## 4.2. 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。

/\*  
static int div(int a, int b) {  
 return a/b;  
}  
\*/  
import "C"  
import "fmt"  
  
func main() {  
 v := C.div(6, 3)  
 fmt.Println(v)  
}

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。

/\*  
#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：

2 <nil>  
0 invalid argument

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。

## 4.3. 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：

//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。

以上有效特性虽然看似有些无聊，但是通过这些例子我们可以精确掌握CGO代码的边界，可以从更深层次的设计的角度来思考产生这些奇怪特性的原因。

## 4.4. 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。

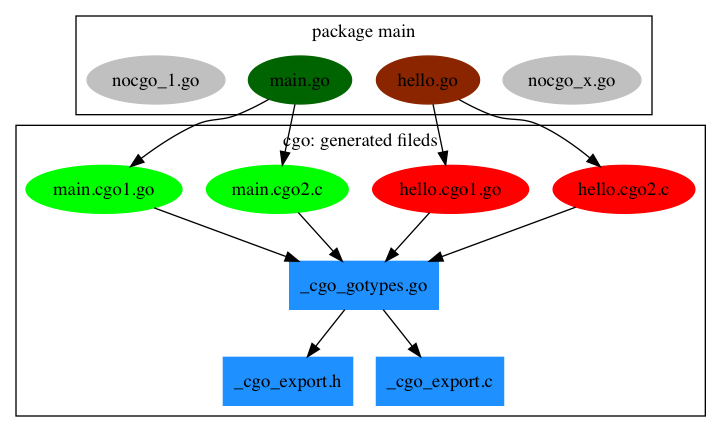
# 5. 内部机制

对于刚刚接触CGO用户来说，CGO的很多特性类似魔法。CGO特性主要是通过一个叫cgo的命令行工具来辅助输出Go和C之间的桥接代码。本节我们尝试从生成的代码分析Go语言和C语言函数直接相互调用的流程。

## 5.1. CGO生成的中间文件

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 2-4 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。

## 5.2. Go调用C函数

Go调用C函数是CGO最常见的应用场景，我们将从最简单的例子入手分析Go调用C函数的详细流程。

具体代码如下（main.go）：

package main  
  
//int sum(int a, int b) { return a+b; }  
import "C"  
  
func main() {  
 println(C.sum(1, 1))  
}

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：

//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：

import "unsafe"  
  
//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];

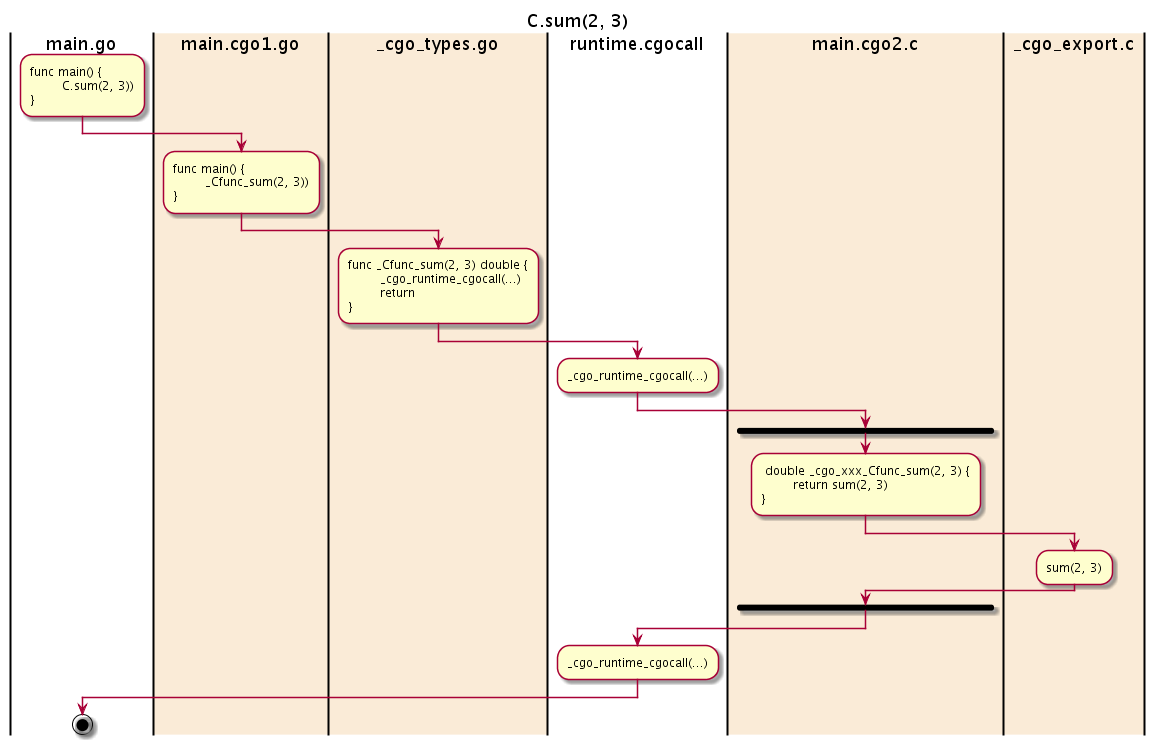
} \_\_attribute\_\_((\_\_packed\_\_)) \*a = v;

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](https://golang.org/src/cmd/cgo/doc.go" \t "http://books.studygolang.com/advanced-go-programming-book/ch2-cgo/_blank) 内部的代码注释和 runtime.cgocall 函数的实现。

## 5.3. 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：

//int sum(int a, int b);  
import "C"  
  
//***export*** sum  
func sum(a, b C.int) C.int {  
 return a + b  
}  
  
func main() {}

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：

$ go build -buildmode=c-archive -o sum.a sum.go

如果没有错误的话，以上编译命令将生成一个sum.a静态库和sum.h头文件。其中sum.h头文件将包含sum函数的声明，静态库中将包含sum函数的实现。

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 main.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)  
}

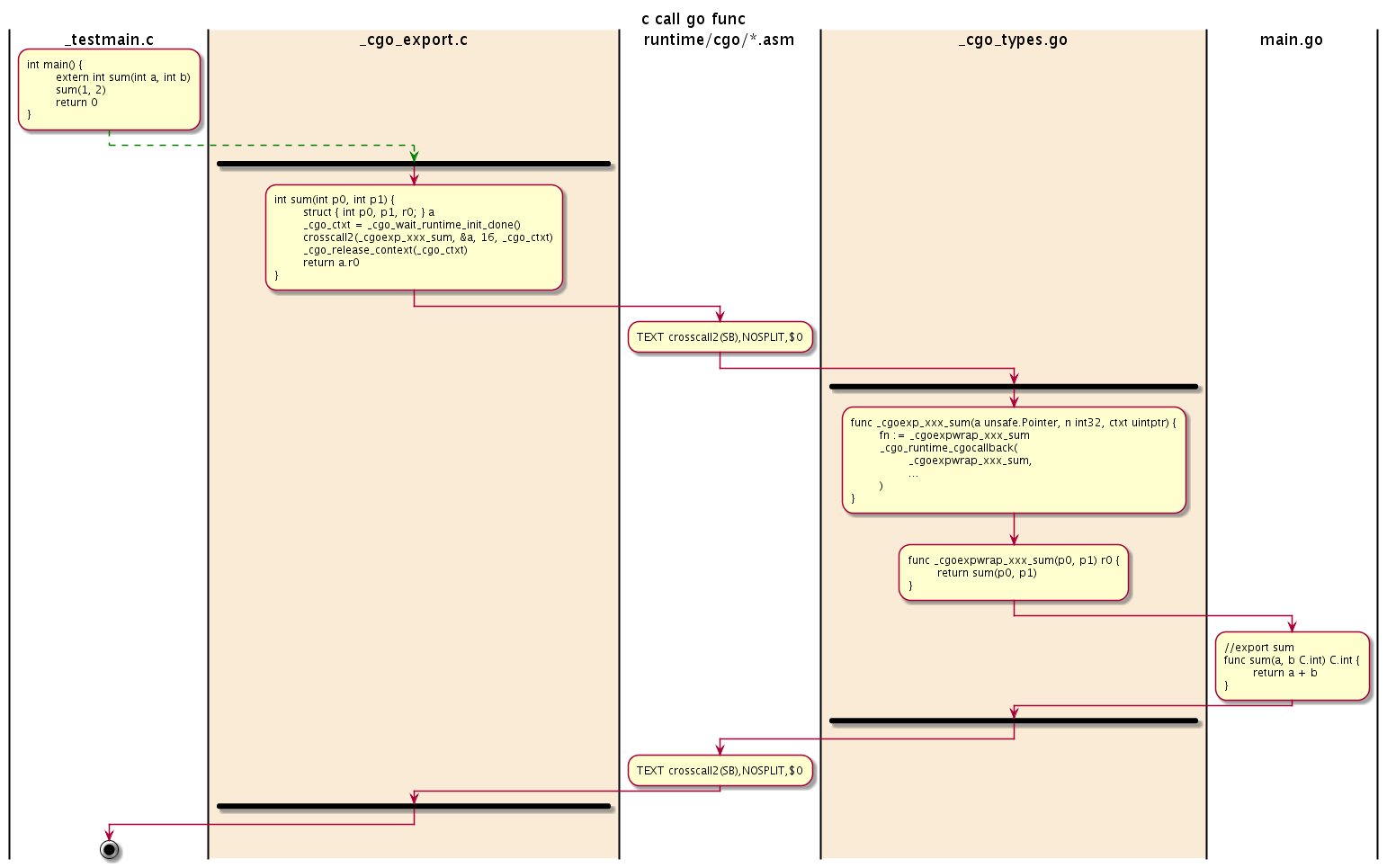
内部将sum的包装函数\_cgoexpwrap\_8313eaf44386\_sum作为函数指针，然后由\_cgo\_runtime\_cgocallback函数完成C语言到Go函数的回调工作。

\_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。

# 6. 实战: 封装qsort

qsort快速排序函数是C语言的高阶函数，支持用于自定义排序比较函数，可以对任意类型的数组进行排序。本节我们尝试基于C语言的qsort函数封装一个Go语言版本的qsort函数。

## **2.6.1 认识qsort函数**

qsort快速排序函数有<stdlib.h>标准库提供，函数的声明如下：

void qsort(  
 void\* base, size\_t num, size\_t size,  
 int (\*cmp)(const void\*, const void\*)  
);

其中base参数是要排序数组的首个元素的地址，num是数组中元素的个数，size是数组中每个元素的大小。最关键是cmp比较函数，用于对数组中任意两个元素进行排序。cmp排序函数的两个指针参数分别是要比较的两个元素的地址，如果第一个参数对应元素大于第二个参数对应的元素将返回结果大于0，如果两个元素相等则返回0，如果第一个元素小于第二个元素则返回结果小于0。

下面的例子是用C语言的qsort对一个int类型的数组进行排序：

#include <stdio.h>

#include <stdlib.h>

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

static int cmp(const void\* a, const void\* b) {

const int\* pa = (int\*)a;

const int\* pb = (int\*)b;

return \*pa - \*pb;

}

int main() {

int values[] = { 42, 8, 109, 97, 23, 25 };

int i;

qsort(values, DIM(values), sizeof(values[0]), cmp);

for(i = 0; i < DIM(values); i++) {

printf ("%d ",values[i]);

}

return 0;

}

其中DIM(values)宏用于计算数组元素的个数，sizeof(values[0])用于计算数组元素的大小。 cmp是用于排序时比较两个元素大小的回调函数。为了避免对全局名字空间的污染，我们将cmp回调函数定义为仅当前文件内可访问的静态函数。

## 6.2. 将qsort函数从Go包导出

为了方便Go语言的非CGO用户使用qsort函数，我们需要将C语言的qsort函数包装为一个外部可以访问的Go函数。

用Go语言将qsort函数重新包装为qsort.Sort函数：

package qsort  
  
//typedef int (\*qsort\_cmp\_func\_t)(const void\* a, const void\* b);  
import "C"  
import "unsafe"  
  
func Sort(  
 base unsafe.Pointer, num, size C.size\_t,  
 cmp C.qsort\_cmp\_func\_t,  
) {  
 C.qsort(base, num, size, cmp)  
}

因为Go语言的CGO语言不好直接表达C语言的函数类型，因此在C语言空间将比较函数类型重新定义为一个qsort\_cmp\_func\_t类型。

虽然Sort函数已经导出了，但是对于qsort包之外的用户依然不能直接使用该函数——Sort函数的参数还包含了虚拟的C包提供的类型。 在CGO的内部机制一节中我们已经提过，虚拟的C包下的任何名称其实都会被映射为包内的私有名字。比如C.size\_t会被展开为\_Ctype\_size\_t，C.qsort\_cmp\_func\_t类型会被展开为\_Ctype\_qsort\_cmp\_func\_t。

被CGO处理后的Sort函数的类型如下：

func Sort(  
 base unsafe.Pointer, num, size \_Ctype\_size\_t,  
 cmp \_Ctype\_qsort\_cmp\_func\_t,  
)

这样将会导致包外部用于无法构造\_Ctype\_size\_t和\_Ctype\_qsort\_cmp\_func\_t类型的参数而无法使用Sort函数。因此，导出的Sort函数的参数和返回值要避免对虚拟C包的依赖。

重新调整Sort函数的参数类型和实现如下：

/\*  
#include <stdlib.h>  
  
typedef int (\*qsort\_cmp\_func\_t)(const void\* a, const void\* b);  
\*/  
import "C"  
import "unsafe"  
  
type CompareFunc C.qsort\_cmp\_func\_t  
  
func Sort(base unsafe.Pointer, num, size int, cmp CompareFunc) {  
 C.qsort(base, C.size\_t(num), C.size\_t(size), C.qsort\_cmp\_func\_t(cmp))  
}

我们将虚拟C包中的类型通过Go语言类型代替，在内部调用C函数时重新转型为C函数需要的类型。因此外部用户将不再依赖qsort包内的虚拟C包。

以下代码展示的Sort函数的使用方式：

package main  
  
//extern int go\_qsort\_compare(void\* a, void\* b);  
import "C"  
  
import (  
 "fmt"  
 "unsafe"  
  
 qsort "."  
)  
  
//***export*** go\_qsort\_compare  
func go\_qsort\_compare(a, b unsafe.Pointer) C.int {  
 pa, pb := (\*C.int)(a), (\*C.int)(b)  
 return C.int(\*pa - \*pb)  
}  
  
func main() {  
 values := []int32{42, 9, 101, 95, 27, 25}  
  
 qsort.Sort(unsafe.Pointer(&values[0]),  
 len(values), int(unsafe.Sizeof(values[0])),  
 qsort.CompareFunc(C.go\_qsort\_compare),  
 )  
 fmt.Println(values)  
}

为了使用Sort函数，我们需要将Go语言的切片取首地址、元素个数、元素大小等信息作为调用参数，同时还需要提供一个C语言规格的比较函数。 其中go\_qsort\_compare是用Go语言实现的，并导出到C语言空间的函数，用于qsort排序时的比较函数。

目前已经实现了对C语言的qsort初步包装，并且可以通过包的方式被其它用户使用。但是qsort.Sort函数已经有很多不便使用之处：用户要提供C语言的比较函数，这对许多Go语言用户是一个挑战。下一步我们将继续改进qsort函数的包装函数，尝试通过闭包函数代替C语言的比较函数。

消除用户对CGO代码的直接依赖。

## 6.3. 改进：闭包函数作为比较函数

在改进之前我们先回顾下Go语言sort包自带的排序函数的接口：

func Slice(slice interface{}, less func(i, j int) bool)

标准库的sort.Slice因为支持通过闭包函数指定比较函数，对切片的排序非常简单：

import (  
 "fmt"  
 "sort"  
)  
  
func main() {  
 values := []int32{42, 9, 101, 95, 27, 25}  
  
 sort.Slice(values, func(i, j int) bool {  
 return values[i] < values[j]  
 })  
  
 fmt.Println(values)  
}

我们也尝试将C语言的qsort函数包装为以下格式的Go语言函数：

package qsort

func Sort(base unsafe.Pointer, num, size int, cmp func(a, b unsafe.Pointer) int)

闭包函数无法导出为C语言函数，因此无法直接将闭包函数传入C语言的qsort函数。 为此我们可以用Go构造一个可以导出为C语言的代理函数，然后通过一个全局变量临时保存当前的闭包比较函数。

代码如下：

var go\_qsort\_compare\_info struct {

fn func(a, b unsafe.Pointer) int

sync.Mutex

}

//export \_cgo\_qsort\_comparefunc \_cgo\_qsort\_compare(a, b unsafe.Pointer) C.int {

return C.int(go\_qsort\_compare\_info.fn(a, b))

}

其中导出的C语言函数\_cgo\_qsort\_compare是公用的qsort比较函数，内部通过go\_qsort\_compare\_info.fn来调用当前的闭包比较函数。

新的Sort包装函数实现如下：

/\*

#include <stdlib.h>

typedef int (\*qsort\_cmp\_func\_t)(const void\* a, const void\* b);

extern int \_cgo\_qsort\_compare(void\* a, void\* b);

\*/import "C"

func Sort(base unsafe.Pointer, num, size int, cmp func(a, b unsafe.Pointer) int) {

go\_qsort\_compare\_info.Lock()

defer go\_qsort\_compare\_info.Unlock()

go\_qsort\_compare\_info.fn = cmp

C.qsort(base, C.size\_t(num), C.size\_t(size),

C.qsort\_cmp\_func\_t(C.\_cgo\_qsort\_compare),

)

}

每次排序前，对全局的go\_qsort\_compare\_info变量加锁，同时将当前的闭包函数保存到全局变量，然后调用C语言的qsort函数。

基于新包装的函数，我们可以简化之前的排序代码：

import (  
 "fmt"  
 "unsafe"  
)  
  
func main() {  
 values := []int32{42, 9, 101, 95, 27, 25}  
  
 qsort.Sort(unsafe.Pointer(&values[0]), len(values), int(unsafe.Sizeof(values[0])),  
 func(a, b unsafe.Pointer) int {  
 pa, pb := (\*int32)(a), (\*int32)(b)  
 return int(\*pa - \*pb)  
 },  
 )  
  
 fmt.Println(values)  
}

现在排序不再需要通过CGO实现C语言版本的比较函数了，可以传入Go语言闭包函数作为比较函数。 但是导入的排序函数依然依赖unsafe包，这是违背Go语言编程习惯的。

## 6.4. 改进：消除用户对unsafe包的依赖

前一个版本的qsort.Sort包装函数已经比最初的C语言版本的qsort易用很多，但是依然保留了很多C语言底层数据结构的细节。 现在我们将继续改进包装函数，尝试消除对unsafe包的依赖，并实现一个类似标准库中sort.Slice的排序函数。

新的包装函数声明如下：

package qsort

func Slice(slice interface{}, less func(a, b int) bool)

首先，我们将slice作为接口类型参数传入，这样可以适配不同的切片类型。 然后切片的首个元素的地址、元素个数和元素大小可以通过reflect反射包从切片中获取。

为了保存必要的排序上下文信息，我们需要在全局包变量增加要排序数组的地址、元素个数和元素大小等信息，比较函数改为less：

var go\_qsort\_compare\_info struct {

base unsafe.Pointer

elemnum int

elemsize int

less func(a, b int) bool

sync.Mutex

}

同样比较函数需要根据元素指针、排序数组的开始地址和元素的大小计算出元素对应数组的索引下标， 然后根据less函数的比较结果返回qsort函数需要格式的比较结果。

//***export*** \_cgo\_qsort\_compare  
func \_cgo\_qsort\_compare(a, b unsafe.Pointer) C.int {  
 var (  
 // array memory is locked  
 base = uintptr(go\_qsort\_compare\_info.base)  
 elemsize = uintptr(go\_qsort\_compare\_info.elemsize)  
 )  
  
 i := int((uintptr(a) - base) / elemsize)  
 j := int((uintptr(b) - base) / elemsize)  
  
 switch {  
 case go\_qsort\_compare\_info.less(i, j): // v[i] < v[j]  
 return -1  
 case go\_qsort\_compare\_info.less(j, i): // v[i] > v[j]  
 return +1  
 default:  
 return 0  
 }  
}

新的Slice函数的实现如下：

import "C"  
import (  
 "fmt"  
 "reflect"  
 "unsafe"  
)  
func Slice(slice interface{}, less func(a, b int) bool) {  
 sv := reflect.ValueOf(slice)  
 if sv.Kind() != reflect.*Slice* {  
 panic(fmt.Sprintf("qsort called with non-slice value of type %T", slice))  
 }  
 if sv.Len() == 0 {  
 return  
 }  
  
 go\_qsort\_compare\_info.Lock()  
 defer go\_qsort\_compare\_info.Unlock()  
  
 defer func() {  
 go\_qsort\_compare\_info.base = nil  
 go\_qsort\_compare\_info.elemnum = 0  
 go\_qsort\_compare\_info.elemsize = 0  
 go\_qsort\_compare\_info.less = nil  
 }()  
  
 // baseMem = unsafe.Pointer(sv.Index(0).Addr().Pointer())  
 // baseMem maybe moved, so must saved after call C.fn  
 go\_qsort\_compare\_info.base = unsafe.Pointer(sv.Index(0).Addr().Pointer())  
 go\_qsort\_compare\_info.elemnum = sv.Len()  
 go\_qsort\_compare\_info.elemsize = int(sv.Type().Elem().Size())  
 go\_qsort\_compare\_info.less = less  
  
 C.qsort(  
 go\_qsort\_compare\_info.base,  
 C.size\_t(go\_qsort\_compare\_info.elemnum),  
 C.size\_t(go\_qsort\_compare\_info.elemsize),  
 C.qsort\_cmp\_func\_t(C.\_cgo\_qsort\_compare),  
 )  
}

首先需要判断传入的接口类型必须是切片类型。然后通过反射获取qsort函数需要的切片信息，并调用C语言的qsort函数。

基于新包装的函数我们可以采用和标准库相似的方式排序切片：

import (  
 "fmt"  
  
 qsort "."  
)  
  
func main() {  
 values := []int64{42, 9, 101, 95, 27, 25}  
  
 qsort.Slice(values, func(i, j int) bool {  
 return values[i] < values[j]  
 })  
  
 fmt.Println(values)  
}

为了避免在排序过程中，排序数组的上下文信息go\_qsort\_compare\_info被修改，我们进行了全局加锁。 因此目前版本的qsort.Slice函数是无法并发执行的，读者可以自己尝试改进这个限制。

# 7. CGO内存模型

CGO是架接Go语言和C语言的桥梁，它使二者在二进制接口层面实现了互通，但是我们要注意因两种语言的内存模型的差异而可能引起的问题。如果在CGO处理的跨语言函数调用时涉及到了指针的传递，则可能会出现Go语言和C语言共享某一段内存的场景。我们知道C语言的内存在分配之后就是稳定的，但是Go语言因为函数栈的动态伸缩可能导致栈中内存地址的移动(这是Go和C内存模型的最大差异)。如果C语言持有的是移动之前的Go指针，那么以旧指针访问Go对象时会导致程序崩溃。

## 7.1. Go访问C内存

C语言空间的内存是稳定的，只要不是被人为提前释放，那么在Go语言空间可以放心大胆地使用。在Go语言访问C语言内存是最简单的情形，我们在之前的例子中已经见过多次。

因为Go语言实现的限制，我们无法在Go语言中创建大于2GB内存的切片（具体请参考makeslice实现代码）。不过借助cgo技术，我们可以在C语言环境创建大于2GB的内存，然后转为Go语言的切片使用：

package main  
  
/\*  
#include <stdlib.h>  
  
void\* makeslice(size\_t memsize) {  
 return malloc(memsize);  
}  
\*/  
import "C"  
import "unsafe"  
  
func makeByteSlize(n int) []byte {  
 p := C.makeslice(C.size\_t(n))  
 return ((\*[1 << 31]byte)(p))[0:n:n]  
}  
  
func freeByteSlice(p []byte) {  
 C.free(unsafe.Pointer(&p[0]))  
}  
  
func main() {  
 s := makeByteSlize(1<<32+1)  
 s[len(s)-1] = 255  
 print(s[len(s)-1])  
 freeByteSlice(s)  
}

例子中我们通过makeByteSlize来创建大于4G内存大小的切片，从而绕过了Go语言实现的限制（需要代码验证）。而freeByteSlice辅助函数则用于释放从C语言函数创建的切片。

因为C语言内存空间是稳定的，基于C语言内存构造的切片也是绝对稳定的，不会因为Go语言栈的变化而被移动。

## 7.2. C临时访问传入的Go内存

cgo之所以存在的一大因素是为了方便在Go语言中接纳吸收过去几十年来使用C/C++语言软件构建的大量的软件资源。C/C++很多库都是需要通过指针直接处理传入的内存数据的，因此cgo中也有很多需要将Go内存传入C语言函数的应用场景。

假设一个极端场景：我们将一块位于某goroutinue的栈上的Go语言内存传入了C语言函数后，在此C语言函数执行期间，此goroutinue的栈因为空间不足的原因发生了扩展，也就是导致了原来的Go语言内存被移动到了新的位置。但是此时此刻C语言函数并不知道该Go语言内存已经移动了位置，仍然用之前的地址来操作该内存——这将将导致内存越界。以上是一个推论（真实情况有些差异），也就是说C访问传入的Go内存可能是不安全的！

当然有RPC远程过程调用的经验的用户可能会考虑通过完全传值的方式处理：借助C语言内存稳定的特性，在C语言空间先开辟同样大小的内存，然后将Go的内存填充到C的内存空间；返回的内存也是如此处理。下面的例子是这种思路的具体实现：

/\*  
void printString(const char\* s) {  
 printf("%s", s);  
}  
\*/  
import "C"  
import "unsafe"  
  
func printString(s string) {  
 cs := C.CString(s)  
 defer C.free(unsafe.Pointer(cs))  
  
 C.printString(cs)  
}  
  
func main() {  
 s := "hello"  
 printString(s)  
}

在需要将Go的字符串传入C语言时，先通过C.CString将Go语言字符串对应的内存数据复制到新创建的C语言内存空间上。上面例子的处理思路虽然是安全的，但是效率极其低下（因为要多次分配内存并逐个复制元素），同时也极其繁琐。

为了简化并高效处理此种向C语言传入Go语言内存的问题，cgo针对该场景定义了专门的规则：在CGO调用的C语言函数返回前，cgo保证传入的Go语言内存在此期间不会发生移动，C语言函数可以大胆地使用Go语言的内存！

根据新的规则我们可以直接传入Go字符串的内存：

package main  
  
/\*  
#include<stdio.h>  
  
void printString(const char\* s, int n) {  
 int i;  
 for(i = 0; i < n; i++) {  
 putchar(s[i]);  
 }  
 putchar('\n');  
}  
\*/  
import "C"  
import (  
 "reflect"  
 "unsafe"  
)  
  
func printString(s string) {  
 p := (\*reflect.StringHeader)(unsafe.Pointer(&s))  
 C.printString((\*C.char)(unsafe.Pointer(p.Data)), C.int(len(s)))  
}  
  
func main() {  
 s := "hello"  
 printString(s)  
}

现在的处理方式更加直接，且避免了分配额外的内存。完美的解决方案！

任何完美的技术都有被滥用的时候，CGO的这种看似完美的规则也是存在隐患的。我们假设调用的C语言函数需要长时间运行，那么将会导致被他引用的Go语言内存在C语言返回前不能被移动，从而可能间接地导致这个Go内存栈对应的goroutine不能动态伸缩栈内存，也就是可能导致这个goroutine被阻塞。因此，在需要长时间运行的C语言函数（特别是在纯CPU运算之外，还可能因为需要等待其它的资源而需要不确定时间才能完成的函数），需要谨慎处理传入的Go语言内存。

不过需要小心的是在取得Go内存后需要马上传入C语言函数，不能保存到临时变量后再间接传入C语言函数。因为CGO只能保证在C函数调用之后被传入的Go语言内存不会发生移动，它并不能保证在传入C函数之前内存不发生变化。

以下代码是错误的：

// 错误的代码  
tmp := uintptr(unsafe.Pointer(&x))  
pb := (\*int16)(unsafe.Pointer(tmp))  
\*pb = 42

因为tmp并不是指针类型，在它获取到Go对象地址之后x对象可能会被移动，但是因为不是指针类型，所以不会被Go语言运行时更新成新内存的地址。在非指针类型的tmp保持Go对象的地址，和在C语言环境保持Go对象的地址的效果是一样的：如果原始的Go对象内存发生了移动，Go语言运行时并不会同步更新它们。

## 7.3. C长期持有Go指针对象

作为一个Go程序员在使用CGO时潜意识会认为总是Go调用C函数。其实CGO中，C语言函数也可以回调Go语言实现的函数。特别是我们可以用Go语言写一个动态库，导出C语言规范的接口给其它用户调用。当C语言函数调用Go语言函数的时候，C语言函数就成了程序的调用方，Go语言函数返回的Go对象内存的生命周期也就自然超出了Go语言运行时的管理。简言之，我们不能在C语言函数中直接使用Go语言对象的内存。

虽然Go语言禁止在C语言函数中长期持有Go指针对象，但是这种需求是切实存在的。如果需要在C语言中访问Go语言内存对象，我们可以将Go语言内存对象在Go语言空间映射为一个int类型的id，然后通过此id来间接访问和控制Go语言对象。

以下代码用于将Go对象映射为整数类型的ObjectId，用完之后需要手工调用free方法释放该对象ID：

package main  
  
import "sync"  
  
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  
}  
  
func NewObjectId(obj interface{}) ObjectId {  
 refs.Lock()  
 defer refs.Unlock()  
  
 id := refs.next  
 refs.next++  
  
 refs.objs[id] = obj  
 return id  
}  
  
func (id ObjectId) IsNil() bool {  
 return id == 0  
}  
  
func (id ObjectId) Get() interface{} {  
 refs.Lock()  
 defer refs.Unlock()  
  
 return refs.objs[id]  
}  
  
func (id \*ObjectId) Free() interface{} {  
 refs.Lock()  
 defer refs.Unlock()  
  
 obj := refs.objs[\*id]  
 delete(refs.objs, \*id)  
 \*id = 0  
  
 return obj  
}

我们通过一个map来管理Go语言对象和id对象的映射关系。其中NewObjectId用于创建一个和对象绑定的id，而id对象的方法可用于解码出原始的Go对象，也可以用于结束id和原始Go对象的绑定。

下面一组函数以C接口规范导出，可以被C语言函数调用：

package main  
  
/\*  
extern char\* NewGoString(char\* );  
extern void FreeGoString(char\* );  
extern void PrintGoString(char\* );  
  
static void printString(const char\* s) {  
 char\* gs = NewGoString(s);  
 PrintGoString(gs);  
 FreeGoString(gs);  
}  
\*/  
import "C"  
import "unsafe"  
  
//***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()  
}  
  
//***export*** PrintGoString  
func PrintGoString(s \*C.char) {  
 id := ObjectId(uintptr(unsafe.Pointer(p)))  
 gs := id.Get().(string)  
 print(gs)  
}  
  
func main() {  
 C.printString("hello")  
}

在printString函数中，我们通过NewGoString创建一个对应的Go字符串对象，返回的其实是一个id，不能直接使用。我们借助PrintGoString函数将id解析为Go语言字符串后打印。该字符串在C语言函数中完全跨越了Go语言的内存管理，在PrintGoString调用前即使发生了栈伸缩导致的Go字符串地址发生变化也依然可以正常工作，因为该字符串对应的id是稳定的，在Go语言空间通过id解码得到的字符串也就是有效的。

## 7.4. 导出C函数不能返回Go内存

在Go语言中，Go是从一个固定的虚拟地址空间分配内存。而C语言分配的内存则不能使用Go语言保留的虚拟内存空间。在CGO环境，Go语言运行时默认会检查导出返回的内存是否是由Go语言分配的，如果是则会抛出运行时异常。

下面是CGO运行时异常的例子：

/\*  
extern int\* getGoPtr();  
  
static void Main() {  
 int\* p = getGoPtr();  
 \*p = 42;  
}  
\*/  
import "C"  
  
func main() {  
 C.Main()  
}  
  
//***export*** getGoPtr  
func getGoPtr() \*C.int {  
 return new(C.int)  
}

其中getGoPtr返回的虽然是C语言类型的指针，但是内存本身是从Go语言的new函数分配，也就是由Go语言运行时统一管理的内存。然后我们在C语言的Main函数中调用了getGoPtr函数，此时默认将发送运行时异常：

$ go run main.go

panic: runtime error: cgo result has Go pointer

goroutine 1 [running]:

main.\_cgoexpwrap\_cfb3840e3af2\_getGoPtr.func1(0xc420051dc0)

command-line-arguments/\_obj/\_cgo\_gotypes.go:60 +0x3a

main.\_cgoexpwrap\_cfb3840e3af2\_getGoPtr(0xc420016078)

command-line-arguments/\_obj/\_cgo\_gotypes.go:62 +0x67

main.\_Cfunc\_Main()

command-line-arguments/\_obj/\_cgo\_gotypes.go:43 +0x41

main.main()

/Users/chai/go/src/github.com/chai2010 \

/advanced-go-programming-book/examples/ch2-xx \

/return-go-ptr/main.go:17 +0x20

exit status 2

异常说明cgo函数返回的结果中含有Go语言分配的指针。指针的检查操作发生在C语言版的getGoPtr函数中，它是由cgo生成的桥接C语言和Go语言的函数。

下面是cgo生成的C语言版本getGoPtr函数的具体细节（在cgo生成的\_cgo\_export.c文件定义）：

int\* getGoPtr(){

\_\_SIZE\_TYPE\_\_ \_cgo\_ctxt = \_cgo\_wait\_runtime\_init\_done();

struct {

int\* r0;

} \_\_attribute\_\_((\_\_packed\_\_)) a;

\_cgo\_tsan\_release();

crosscall2(\_cgoexp\_95d42b8e6230\_getGoPtr, &a, 8, \_cgo\_ctxt);

\_cgo\_tsan\_acquire();

\_cgo\_release\_context(\_cgo\_ctxt);

return a.r0;

}

其中\_cgo\_tsan\_acquire是从LLVM项目移植过来的内存指针扫描函数，它会检查cgo函数返回的结果是否包含Go指针。

需要说明的是，cgo默认对返回结果的指针的检查是有代价的，特别是cgo函数返回的结果是一个复杂的数据结构时将花费更多的时间。如果已经确保了cgo函数返回的结果是安全的话，可以通过设置环境变量GODEBUG=cgocheck=0来关闭指针检查行为。

$ GODEBUG=cgocheck=0 go run main.go

关闭cgocheck功能后再运行上面的代码就不会出现上面的异常的。但是要注意的是，如果C语言使用期间对应的内存被Go运行时释放了，将会导致更严重的崩溃问题。cgocheck默认的值是1，对应一个简化版本的检测，如果需要完整的检测功能可以将cgocheck设置为2。

关于cgo运行时指针检测的功能详细说明可以参考Go语言的官方文档。

# 8. C++ 类包装

CGO是C语言和Go语言之间的桥梁，原则上无法直接支持C++的类。CGO不支持C++语法的根本原因是C++至今为止还没有一个二进制接口规范(ABI)。一个C++类的构造函数在编译为目标文件时如何生成链接符号名称、方法在不同平台甚至是C++的不同版本之间都是不一样的。但是C++是兼容C语言，所以我们可以通过增加一组C语言函数接口作为C++类和CGO之间的桥梁，这样就可以间接地实现C++和Go之间的互联。当然，因为CGO只支持C语言中值类型的数据类型，所以我们是无法直接使用C++的引用参数等特性的。

## 8.1. C++ 类到 Go 语言对象

实现C++类到Go语言对象的包装需要经过以下几个步骤：首先是用纯C函数接口包装该C++类；其次是通过CGO将纯C函数接口映射到Go函数；最后是做一个Go包装对象，将C++类到方法用Go对象的方法实现。

### 8.1.1. 准备一个 C++ 类

为了演示简单，我们基于std::string做一个最简单的缓存类MyBuffer。除了构造函数和析构函数之外，只有两个成员函数分别是返回底层的数据指针和缓存的大小。因为是二进制缓存，所以我们可以在里面中放置任意数据。

// my\_buffer.h  
#include <string>  
  
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();  
 }  
};

我们在构造函数中指定缓存的大小并分配空间，在使用完之后通过析构函数释放内部分配的内存空间。下面是简单的使用方式：

int main() {  
 auto pBuf = new MyBuffer(1024);  
  
 auto data = pBuf->Data();  
 auto size = pBuf->Size();  
  
 delete pBuf;  
}

为了方便向C语言接口过渡，在此处我们故意没有定义C++的拷贝构造函数。我们必须以new和delete来分配和释放缓存对象，而不能以值风格的方式来使用。

### 8.1.2 用纯C函数接口封装 C++ 类

如果要将上面的C++类用C语言函数接口封装，我们可以从使用方式入手。我们可以将new和delete映射为C语言函数，将对象的方法也映射为C语言函数。

在C语言中我们期望MyBuffer类可以这样使用：

int main() {

MyBuffer\* pBuf = NewMyBuffer(1024);

char\* data = MyBuffer\_Data(pBuf);

auto size = MyBuffer\_Size(pBuf);

DeleteMyBuffer(pBuf);

}

先从C语言接口用户的角度思考需要什么样的接口，然后创建 my\_buffer\_capi.h 头文件接口规范：

// my\_buffer\_capi.htypedef 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);

然后就可以基于C++的MyBuffer类定义这些C语言包装函数。我们创建对应的my\_buffer\_capi.cc文件如下：

// my\_buffer\_capi.cc

#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;

}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();

}

因为头文件my\_buffer\_capi.h是用于CGO，必须是采用C语言规范的名字修饰规则。在C++源文件包含时需要用extern "C"语句说明。另外MyBuffer\_T的实现只是从MyBuffer继承的类，这样可以简化包装代码的实现。同时和CGO通信时必须通过MyBuffer\_T指针，我们无法将具体的实现暴露给CGO，因为实现中包含了C++特有的语法，CGO无法识别C++特性。

将C++类包装为纯C接口之后，下一步的工作就是将C函数转为Go函数。

### 8.1.3. 将纯C接口函数转为Go函数

将纯C函数包装为对应的Go函数的过程比较简单。需要注意的是，因为我们的包中包含C++11的语法，因此需要通过#cgo CXXFLAGS: -std=c++11打开C++11的选项。

// my\_buffer\_capi.go  
  
package main  
  
/\*  
#cgo CXXFLAGS: -std=c++11  
  
#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))  
}  
  
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))  
}

为了区分，我们在Go中的每个类型和函数名称前面增加了cgo\_前缀，比如cgo\_MyBuffer\_T是对应C中的MyBuffer\_T类型。

为了处理简单，在包装纯C函数到Go函数时，除了cgo\_MyBuffer\_T类型外，对输入参数和返回值的基础类型，我们依然是用的C语言的类型。

### 8.1.4. 包装为Go对象

在将纯C接口包装为Go函数之后，我们就可以很容易地基于包装的Go函数构造出Go对象来。因为cgo\_MyBuffer\_T是从C语言空间导入的类型，它无法定义自己的方法，因此我们构造了一个新的MyBuffer类型，里面的成员持有cgo\_MyBuffer\_T指向的C语言缓存对象。

// my\_buffer.go  
  
package main  
  
import "unsafe"  
  
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)  
}  
  
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):int(size)]  
}

同时，因为Go语言的切片本身含有长度信息，我们将cgo\_MyBuffer\_Data和cgo\_MyBuffer\_Size两个函数合并为MyBuffer.Data方法，它返回一个对应底层C语言缓存空间的切片。

现在我们就可以很容易在Go语言中使用包装后的缓存对象了（底层是基于C++的std::string实现）：

package main  
  
//#include <stdio.h>  
import "C"  
import "unsafe"  
  
func main() {  
 buf := NewMyBuffer(1024)  
 defer buf.Delete()  
  
 copy(buf.Data(), []byte("hello\x00"))  
 C.puts((\*C.char)(unsafe.Pointer(&(buf.Data()[0]))))  
}

例子中，我们创建了一个1024字节大小的缓存，然后通过copy函数向缓存填充了一个字符串。为了方便C语言字符串函数处理，我们在填充字符串的默认用'\0'表示字符串结束。最后我们直接获取缓存的底层数据指针，用C语言的puts函数打印缓存的内容。

## 8.2. Go 语言对象到 C++ 类

要实现Go语言对象到C++类的包装需要经过以下几个步骤：首先是将Go对象映射为一个id；然后基于id导出对应的C接口函数；最后是基于C接口函数包装为C++对象。

### 8.2.1 构造一个Go对象

为了便于演示，我们用Go语言构建了一个Person对象，每个Person可以有名字和年龄信息：

package main  
  
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 = name  
 p.age = age  
}  
  
func (p \*Person) Get() (name string, age int) {  
 return p.name, p.age  
}

Person对象如果想要在C/C++中访问，需要通过cgo导出C接口来访问。

### 8.2.2 导出C接口

我们前面仿照C++对象到C接口的过程，也抽象一组C接口描述Person对象。创建一个person\_capi.h文件，对应C接口规范文件：

// 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);

然后是在Go语言中实现这一组C函数。

需要注意的是，通过CGO导出C函数时，输入参数和返回值类型都不支持const修饰，同时也不支持可变参数的函数类型。同时如内存模式一节所述，我们无法在C/C++中直接长期访问Go内存对象。因此我们使用前一节所讲述的技术将Go对象映射为一个整数id。

下面是person\_capi.go文件，对应C接口函数的实现：

// person\_capi.go  
package main  
  
//#include "./person\_capi.h"  
import "C"  
import "unsafe"  
  
//***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))  
}  
  
//***export*** person\_get\_name  
func person\_get\_name(h C.person\_handle\_t, buf \*C.char, size C.int) \*C.char {  
 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  
}  
  
//***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)  
}

在创建Go对象后，我们通过NewObjectId将Go对应映射为id。然后将id强制转义为person\_handle\_t类型返回。其它的接口函数则是根据person\_handle\_t所表示的id，让根据id解析出对应的Go对象。

### 8.2.3. 封装C++对象

有了C接口之后封装C++对象就比较简单了。常见的做法是新建一个Person类，里面包含一个person\_handle\_t类型的成员对应真实的Go对象，然后在Person类的构造函数中通过C接口创建Go对象，在析构函数中通过C接口释放Go对象。下面是采用这种技术的实现：

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\_);  
 }  
  
 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\_);  
 }  
}

包装后我们就可以像普通C++类那样使用了：

#include "person.h"  
  
#include <stdio.h>  
  
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;  
}

### 8.2.4 封装C++对象改进

在前面的封装C++对象的实现中，每次通过new创建一个Person实例需要进行两次内存分配：一次是针对C++版本的Person，再一次是针对Go语言版本的Person。其实C++版本的Person内部只有一个person\_handle\_t类型的id，用于映射Go对象。我们完全可以将person\_handle\_t直接当中C++对象来使用。

下面时改进后的包装方式：

extern "C" {

#include "./person\_capi.h"

}

struct 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, size);

}

int GetAge() {

return person\_get\_age(person\_handle\_t(this));

}

};

我们在Person类中增加了一个叫New静态成员函数，用于创建新的Person实例。在New函数中通过调用person\_new来创建Person实例，返回的是person\_handle\_t类型的id，我们将其强制转型作为Person\*类型指针返回。在其它的成员函数中，我们通过将this指针再反向转型为person\_handle\_t类型，然后通过C接口调用对应的函数。

到此，我们就达到了将Go对象导出为C接口，然后基于C接口再包装为C++对象以便于使用的目的。

## 8.3. 彻底解放C++的this指针

熟悉Go语言的用法会发现Go语言中方法是绑定到类型的。比如我们基于int定义一个新的Int类型，就可以有自己的方法：

package main  
  
import "fmt"  
  
type Int int  
  
func (p Int) Twice() int {  
 return int(p)\*2  
}  
  
func main() {  
 var x = Int(42)  
 fmt.Println(int(x))  
 fmt.Println(x.Twice())  
}

这样就可以在不改变原有数据底层内存结构的前提下，自由切换int和Int类型来使用变量。

而在C++中要实现类似的特性，一般会采用以下实现：

class Int {

int v\_;

Int(v int) { this.v\_ = v; }

int Twice() const{ return this.v\_\*2; }

};

int main() {

Int v(42);

printf("%d\n", v); // error

printf("%d\n", v.Twice());

}

新包装后的Int类虽然增加了Twice方法，但是失去了自由转回int类型的权利。这时候不仅连printf都无法输出Int本身的值，而且也失去了int类型运算的所有特性。这就是C++构造函数的邪恶之处：以失去原有的一切特性的代价换取class的施舍。

造成这个问题的根源是C++中this被固定为class的指针类型了。我们重新回顾下this在Go语言中的本质：

func (this Int) Twice() intfunc Int\_Twice(this Int) int

在Go语言中，和this有着相似功能的类型接收者参数其实只是一个普通的函数参数，我们可以自由选择值或指针类型。

如果以C语言的角度来思考，this也只是一个普通的void\*类型的指针，我们可以随意自由地将this转换为其它类型。

struct Int {

int Twice() {

const int\* p = (int\*)(this);

return (\*p) \* 2;

}

};

int main() {

int x = 42;

printf("%d\n", x);

printf("%d\n", ((Int\*)(&x))->Twice());

return 0;

}

这样我们就可以通过将int类型指针强制转为Int类型指针，代替通过默认的构造函数后new来构造Int对象。 在Twice函数的内部，以相反的操作将this指针转回int类型的指针，就可以解析出原有的int类型的值了。 这时候Int类型只是编译时的一个壳子，并不会在运行时占用额外的空间。

因此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可以绑定到类型的。 只有纯虚方法是绑定到对象，那就是接口。

# 9 静态库和动态库

CGO在使用C/C++资源的时候一般有三种形式：直接使用源码；链接静态库；链接动态库。直接使用源码就是在import "C"之前的注释部分包含C代码，或者在当前包中包含C/C++源文件。链接静态库和动态库的方式比较类似，都是通过在LDFLAGS选项指定要链接的库方式链接。本节我们主要关注在CGO中如何使用静态库和动态库相关的问题。

## 9.1. 使用C静态库

如果CGO中引入的C/C++资源有代码而且代码规模也比较小，直接使用源码是最理想的方式，但很多时候我们并没有源代码，或者从C/C++源代码开始构建的过程异常复杂，这种时候使用C静态库也是一个不错的选择。静态库因为是静态链接，最终的目标程序并不会产生额外的运行时依赖，也不会出现动态库特有的跨运行时资源管理的错误。不过静态库对链接阶段会有一定要求：静态库一般包含了全部的代码，里面会有大量的符号，如果不同静态库之间出现了符号冲突则会导致链接的失败。

我们先用纯C语言构造一个简单的静态库。我们要构造的静态库名叫number，库中只有一个number\_add\_mod函数，用于表示数论中的模加法运算。number库的文件都在number目录下。

number/number.h头文件只有一个纯C语言风格的函数声明：

int number\_add\_mod(int a, int b, int mod);

number/number.c对应函数的实现：

#include "number.h"

int number\_add\_mod(int a, int b, int mod) {

return (a+b)%mod;

}

因为CGO使用的是GCC命令来编译和链接C和Go桥接的代码。因此静态库也必须是GCC兼容的格式。

通过以下命令可以生成一个叫libnumber.a的静态库：

$ cd ./number

$ gcc -c -o number.o number.c

$ ar rcs libnumber.a number.o

生成libnumber.a静态库之后，我们就可以在CGO中使用该资源了。

创建main.go文件如下：

package main  
  
//#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))  
}

其中有两个#cgo命令，分别是编译和链接参数。CFLAGS通过-I./number将number库对应头文件所在的目录加入头文件检索路径。LDFLAGS通过-L${SRCDIR}/number将编译后number静态库所在目录加为链接库检索路径，-lnumber表示链接libnumber.a静态库。需要注意的是，在链接部分的检索路径不能使用相对路径（C/C++代码的链接程序所限制），我们必须通过cgo特有的${SRCDIR}变量将源文件对应的当前目录路径展开为绝对路径（因此在windows平台中绝对路径不能有空白符号）。

因为我们有number库的全部代码，所以我们可以用go generate工具来生成静态库，或者是通过Makefile来构建静态库。因此发布CGO源码包时，我们并不需要提前构建C静态库。

因为多了一个静态库的构建步骤，这种使用了自定义静态库并已经包含了静态库全部代码的Go包无法直接用go get安装。不过我们依然可以通过go get下载，然后用go generate触发静态库构建，最后才是go install来完成安装。

为了支持go get命令直接下载并安装，我们C语言的#include语法可以将number库的源文件链接到当前的包。

创建z\_link\_number\_c.c文件如下：

#include "./number/number.c"

然后在执行go get或go build之类命令的时候，CGO就是自动构建number库对应的代码。这种技术是在不改变静态库源代码组织结构的前提下，将静态库转化为了源代码方式引用。这种CGO包是最完美的。

如果使用的是第三方的静态库，我们需要先下载安装静态库到合适的位置。然后在#cgo命令中通过CFLAGS和LDFLAGS来指定头文件和库的位置。对于不同的操作系统甚至同一种操作系统的不同版本来说，这些库的安装路径可能都是不同的，那么如何在代码中指定这些可能变化的参数呢？

在Linux环境，有一个pkg-config命令可以查询要使用某个静态库或动态库时的编译和链接参数。我们可以在#cgo命令中直接使用pkg-config命令来生成编译和链接参数。而且还可以通过PKG\_CONFIG环境变量定制pkg-config命令。因为不同的操作系统对pkg-config命令的支持不尽相同，通过该方式很难兼容不同的操作系统下的构建参数。不过对于Linux等特定的系统，pkg-config命令确实可以简化构建参数的管理。关于pkg-config的使用细节在此我们不深入展开，大家可以自行参考相关文档。

## 9.2 使用C动态库

动态库出现的初衷是对于相同的库，多个进程可以共享同一个，以节省内存和磁盘资源。但是在磁盘和内存已经白菜价的今天，这两个作用已经显得微不足道了，那么除此之外动态库还有哪些存在的价值呢？从库开发角度来说，动态库可以隔离不同动态库之间的关系，减少链接时出现符号冲突的风险。而且对于windows等平台，动态库是跨越VC和GCC不同编译器平台的唯一的可行方式。

对于CGO来说，使用动态库和静态库是一样的，因为动态库也必须要有一个小的静态导出库用于链接动态库（Linux下可以直接链接so文件，但是在Windows下必须为dll创建一个.a文件用于链接）。我们还是以前面的number库为例来说明如何以动态库方式使用。

对于在macOS和Linux系统下的gcc环境，我们可以用以下命令创建number库的的动态库：

$ cd number

$ gcc -shared -o libnumber.so number.c

因为动态库和静态库的基础名称都是libnumber，只是后缀名不同而已。因此Go语言部分的代码和静态库版本完全一样：

package main  
  
//#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))  
}

编译时GCC会自动找到libnumber.a或libnumber.so进行链接。

对于windows平台，我们还可以用VC工具来生成动态库（windows下有一些复杂的C++库只能用VC构建）。我们需要先为number.dll创建一个def文件，用于控制要导出到动态库的符号。

number.def文件的内容如下：

LIBRARY number.dll

EXPORTS

number\_add\_mod

其中第一行的LIBRARY指明动态库的文件名，然后的EXPORTS语句之后是要导出的符号名列表。

现在我们可以用以下命令来创建动态库（需要进入VC对应的x64命令行环境）。

$ cl /c number.c

$ link /DLL /OUT:number.dll number.obj number.def

这时候会为dll同时生成一个number.lib的导出库。但是在CGO中我们无法使用lib格式的链接库。

要生成.a格式的导出库需要通过mingw工具箱中的dlltool命令完成：

$ dlltool -dllname number.dll --def number.def --output-lib libnumber.a

生成了libnumber.a文件之后，就可以通过-lnumber链接参数进行链接了。

需要注意的是，在运行时需要将动态库放到系统能够找到的位置。对于windows来说，可以将动态库和可执行程序放到同一个目录，或者将动态库所在的目录绝对路径添加到PATH环境变量中。对于macOS来说，需要设置DYLD\_LIBRARY\_PATH环境变量。而对于Linux系统来说，需要设置LD\_LIBRARY\_PATH环境变量。

## 9.3. 导出C静态库

CGO不仅可以使用C静态库，也可以将Go实现的函数导出为C静态库。我们现在用Go实现前面的number库的模加法函数。

创建number.go，内容如下：

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  
}

根据CGO文档的要求，我们需要在main包中导出C函数。对于C静态库构建方式来说，会忽略main包中的main函数，只是简单导出C函数。采用以下命令构建：

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

在生成number.a静态库的同时，cgo还会生成一个number.h文件。

number.h文件的内容如下（为了便于显示，内容做了精简）：

#ifdef \_\_cplusplus  
extern "C" {  
#endif  
  
extern int number\_add\_mod(int p0, int p1, int p2);  
  
#ifdef \_\_cplusplus  
}  
#endif

其中extern "C"部分的语法是为了同时适配C和C++两种语言。核心内容是声明了要导出的number\_add\_mod函数。

然后我们创建一个\_test\_main.c的C文件用于测试生成的C静态库（用下划线作为前缀名是让为了让go build构建C静态库时忽略这个文件）：

#include "number.h"

#include <stdio.h>

int main() {

int a = 10;

int b = 5;

int c = 12;

int x = number\_add\_mod(a, b, c);

printf("(%d+%d)%%%d = %d\n", a, b, c, x);

return 0;

}

通过以下命令编译并运行：

$ gcc -o a.out \_test\_main.c number.a

$ ./a.out

使用CGO创建静态库的过程非常简单。

## 9.4. 导出C动态库

CGO导出动态库的过程和静态库类似，只是将构建模式改为c-shared，输出文件名改为number.so而已：

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

\_test\_main.c文件内容不变，然后用以下命令编译并运行：

$ gcc -o a.out \_test\_main.c number.so

$ ./a.out

## **9.5. 导出非main包的函数**

通过go help buildmode命令可以查看C静态库和C动态库的构建说明：

-buildmode=c-archive

Build the listed main package, plus all packages it imports,

into a C archive file. The only callable symbols will be those

functions exported using a cgo //export comment. Requires

exactly one main package to be listed.

-buildmode=c-shared

Build the listed main package, plus all packages it imports,

into a C shared library. The only callable symbols will

be those functions exported using a cgo //export comment.

Requires exactly one main package to be listed.

文档说明导出的C函数必须是在main包导出，然后才能在生成的头文件包含声明的语句。但是很多时候我们可能更希望将不同类型的导出函数组织到不同的Go包中，然后统一导出为一个静态库或动态库。

要实现从是从非main包导出C函数，或者是多个包导出C函数（因为只能有一个main包），我们需要自己提供导出C函数对应的头文件（因为CGO无法为非main包的导出函数生成头文件）。

假设我们先创建一个number子包，用于提供模加法函数：

package number  
  
import "C"  
  
//***export*** number\_add\_mod  
func number\_add\_mod(a, b, mod C.int) C.int {  
 return (a + b) % mod  
}

然后是当前的main包：

package main  
  
import "C"  
  
import (  
 "fmt"  
  
 \_ "./number"  
)  
  
func main() {  
 println("Done")  
}  
  
//***export*** goPrintln  
func goPrintln(s \*C.char) {  
 fmt.Println("goPrintln:", C.GoString(s))  
}

其中我们导入了number子包，在number子包中有导出的C函数number\_add\_mod，同时我们在main包也导出了goPrintln函数。

通过以下命令创建C静态库：

$ go build -buildmode=c-archive -o main.a

这时候在生成main.a静态库的同时，也会生成一个main.h头文件。但是main.h头文件中只有main包中导出的goPrintln函数的声明，并没有number子包导出函数的声明。其实number\_add\_mod函数在生成的C静态库中是存在的，我们可以直接使用。

创建\_test\_main.c测试文件如下：

#include <stdio.h>

void goPrintln(char\*);int number\_add\_mod(int a, int b, int mod);

int main() {

int a = 10;

int b = 5;

int c = 12;

int x = number\_add\_mod(a, b, c);

printf("(%d+%d)%%%d = %d\n", a, b, c, x);

goPrintln("done");

return 0;

}

我们并没有包含CGO自动生成的main.h头文件，而是通过手工方式声明了goPrintln和number\_add\_mod两个导出函数。这样我们就实现了从多个Go包导出C函数了。

# 10. 编译和链接参数

编译和链接参数是每一个C/C++程序员需要经常面对的问题。构建每一个C/C++应用均需要经过编译和链接两个步骤，CGO也是如此。 本节我们将简要讨论CGO中经常用到的编译和链接参数的用法。

## 10.1. 编译参数：CFLAGS/CPPFLAGS/CXXFLAGS

编译参数主要是头文件的检索路径，预定义的宏等参数。理论上来说C和C++是完全独立的两个编程语言，它们可以有着自己独立的编译参数。 但是因为C++语言对C语言做了深度兼容，甚至可以将C++理解为C语言的超集，因此C和C++语言之间又会共享很多编译参数。 因此CGO提供了CFLAGS/CPPFLAGS/CXXFLAGS三种参数，其中CFLAGS对应C语言编译参数(以.c后缀名)、 CPPFLAGS对应C/C++ 代码编译参数(*.c,*.cc,*.cpp,*.cxx)、CXXFLAGS对应纯C++编译参数(*.cc,*.cpp,\*.cxx)。

## 10.2. 链接参数：LDFLAGS

链接参数主要包含要链接库的检索目录和要链接库的名字。因为历史遗留问题，链接库不支持相对路径，我们必须为链接库指定绝对路径。 cgo 中的 ${SRCDIR} 为当前目录的绝对路径。经过编译后的C和C++目标文件格式是一样的，因此LDFLAGS对应C/C++共同的链接参数。

## 10.3. pkg-config

为不同C/C++库提供编译和链接参数是一项非常繁琐的工作，因此cgo提供了对应pkg-config工具的支持。 我们可以通过#cgo pkg-config xxx命令来生成xxx库需要的编译和链接参数，其底层通过调用 pkg-config xxx --cflags生成编译参数，通过pkg-config xxx --libs命令生成链接参数。 需要注意的是pkg-config工具生成的编译和链接参数是C/C++公用的，无法做更细的区分。

pkg-config工具虽然方便，但是有很多非标准的C/C++库并没有实现对其支持。 这时候我们可以手工为pkg-config工具创建对应库的编译和链接参数实现支持。

比如有一个名为xxx的C/C++库，我们可以手工创建/usr/local/lib/pkgconfig/xxx.bc文件：

Name: xxx

Cflags:-I/usr/local/include

Libs:-L/usr/local/lib –lxxx2

其中Name是库的名字，Cflags和Libs行分别对应xxx使用库需要的编译和链接参数。如果bc文件在其它目录， 可以通过PKG\_CONFIG\_PATH环境变量指定pkg-config工具的检索目录。

而对应cgo来说，我们甚至可以通过PKG\_CONFIG 环境变量可指定自定义的pkg-config程序。 如果是自己实现CGO专用的pkg-config程序，只要处理--cflags和--libs两个参数即可。

下面的程序是macos系统下生成Python3的编译和链接参数：

package main  
  
import (  
 "bytes"  
 "fmt"  
 "os"  
 "os/exec"  
)  
  
// py3-config.go  
func main() {  
 for \_, s := range os.Args {  
 if s == "--cflags" {  
 out, \_ := exec.Command("python3-config", "--cflags").CombinedOutput()  
 out = bytes.Replace(out, []byte("-arch"), []byte{}, -1)  
 out = bytes.Replace(out, []byte("i386"), []byte{}, -1)  
 out = bytes.Replace(out, []byte("x86\_64"), []byte{}, -1)  
 fmt.Print(string(out))  
 return  
 }  
 if s == "--libs" {  
 out, \_ := exec.Command("python3-config", "--ldflags").CombinedOutput()  
 fmt.Print(string(out))  
 return  
 }  
 }  
}

然后通过以下命令构建并使用自定义的pkg-config工具：

$ go build -o py3-config py3-config.go

$ PKG\_CONFIG=./py3-config go build -buildmode=c-shared -o gopkg.so main.go

具体的细节可以参考Go实现Python模块章节。

## 10.4. go get 链

在使用go get获取Go语言包的同时会获取包依赖的包。比如A包依赖B包，B包依赖C包，C包依赖D包： pkgA -> pkgB -> pkgC -> pkgD -> ...。再go get获取A包之后会依次线获取BCD包。 如果在获取B包之后构建失败，那么将导致链条的断裂，从而导致A包的构建失败。

链条断裂的原因有很多，其中常见的原因有：

* 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

仔细分析可以发现，失败的原因中和CGO相关的问题占了绝大多数。这并不是偶然现象， 自动化构建C/C++代码一直是一个世界难题，到目前位置也没有出现一个大家认可的统一的C/C++管理工具。

因为用了cgo，比如gcc等构建工具是必须安装的，同时尽量要做到对主流系统的支持。 如果依赖的C/C++包比较小并且有源代码的前提下，可以优先选择从代码构建。

比如github.com/chai2010/webp包通过为每个C/C++源文件在当前包建立关键文件实现零配置依赖：

// z\_libwebp\_src\_dec\_alpha.c

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

因此在编译z\_libwebp\_src\_dec\_alpha.c文件时，会编译libweb原生的代码。 其中的依赖是相对目录，对于不同的平台支持可以保持最大的一致性。

## 10.5. 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，但是真实情况是其它包的Go导出函数也是有效的。 因为导出后的Go函数就可以当作C函数使用，所以必须有效。但是不同包导出的Go函数将在同一个全局的名字空间，因此需要小心避免重名的问题。 如果是从不同的包导出Go函数到C语言空间，那么cgo自动生成的\_cgo\_export.h文件将无法包含全部到处的函数声明， 我们必须通过手写头文件的方式什么导出的全部函数。

# 11 补充说明

CGO是C语言和Go语言混合编程的技术，因此要想熟练地使用CGO需要了解这两门语言。C语言推荐两本书：第一本是C语言之父编写的《C程序设计语言》；第二本是讲述C语言模块化编程的《C语言接口与实现:创建可重用软件的技术》。Go语言推荐官方出版的《The Go Programming Language》和Go语言自带的全部文档和全部代码。

为何要话费巨大的精力学习CGO是一个问题。任何技术和语言都有它自身的优点和不足，Go语言不是银弹，它无法解决全部问题。而通过CGO可以继承C/C++将近半个世纪的软件遗产，通过CGO可以用Go给其它系统写C接口的共享库，通过CGO技术可以让Go语言编写的代码可以很好地融入现有的软件生态——而现在的软件正式建立在C/C++语言之上的。因此说CGO是一个保底的后备技术，它是Go的一个重量级的替补技术，值得任何一个严肃的Go语言开发人员学习。