一、基础概念

FFI

FFI (Foreign Function Interface)翻译过来叫做外部函数接口，是一种机制，允许一种语言去调用另一种语言，通常用于调用二进制动态链接库的上下文中。最早来自于 Common Lisp 的规范，不过我所使用过的绝大多数语言中都有 FFI 的概念/术语存在，比如：Python、Ruby，Haskell、Go、Rust、LuaJIT 等。

在不同的语言中会有不同的实现，比如在 Go 中的 cgo，Python 中的 ctypes，Haskell 中的 CAPI （之前还有一个 ccall）等。

对于 Go 和 Rust 而言，它们的 FFI 需要与 C 语言对象进行通信，而这部分其实是由操作系统根据 API 中的调用约定来完成的。

二、环境准备

2.1 安装RUST

安装rust: https://www.rust-lang.org/zh-CN/tools/install
安装cbindgen: https://github.com/mozilla/cbindgen

2.2 安装GO

安装golang: https://go.dev/doc/install

三、创建Rust工程

3.1 创建cargo工程

```Python
➜  go-rust-demo cargo new --lib rustdemo
     Created library `rustdemo` package

➜  go-rust-demo cd rustdemo
➜  rustdemo git:(master) ✗ ls
Cargo.toml src

➜  rustdemo git:(master) ✗ tree
.
├── Cargo.lock
├── Cargo.toml
└── src
```

```
└── lib.rs

1 directory, 3 files
```

3.2 添加示例代码

```Python
extern crate libc;
use std::ffi::{CStr, CString};

#[no_mangle]
pub extern "C" fn rustdemo(name: *const libc::c_char) -> *const
libc::c_char {
    let cstr_name = unsafe { CStr::from_ptr(name) };
    let mut str_name = cstr_name.to_str().unwrap().to_string();

    println!("Rust got: {:?}", str_name);

    let r_string: &str = " Rust say: Hello Go ";
    str_name.push_str(r_string);
    CString::new(str_name).unwrap().into_raw()
}
```

注意，其中的`#[no_mangle]`，它用于告诉 Rust 编译器：不要乱改函数的名称。

3.3 添加Rust依赖

```Python
[package]
name = "rustdemo"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html

[lib]
```

```
name="rustdemo"
crate-type = ["staticlib","cdylib"]

[dependencies]
libc="0.2.140"
```

其中，`crate-type`表示要编译的二进制库的类型：动态链接库 OR 静态链接库。

3.4 生成头文件

```
Python
➜  rustdemo git:(master) ✗ cbindgen ./src/lib.rs -l c >
./rustdemo.h
```

生成的头文件内容如下：

```
C/C++
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

const char *rustdemo(const char *name);
```

其中，`rustdemo`就是我们在前面的样例中定义的交互函数。
当然，我们也可以尝试手动去生成这个头文件，但是，当交互接口变得很多时，手动添加可能会遗漏部分内容。

此外，我们也可以添加脚本，用于在编译时自动生成c语言头文件，具体用法在cbindgen的文档中，这里就不在展开。

3.5 编译生成链接库

我们在Cargo.toml中添加依赖时，设置了要生成的库的类型`crate-type`同时为动态链接库和静态连接库。无论是动态链接库还是静态链接库，生成它们时都会遇到交叉编译的问题——和平台强相关。这里，我们参考借鉴了以下文档的做法：
https://gist.github.com/surpher/bbf88e191e9d1f01ab2e2bbb85f9b528

在Mac的M1芯片平台上，使用一下命令来生成相关链接库：

```
Python
➜  rustdemo git:(master) ✗ cargo build
--target=aarch64-apple-darwin --release
   Compiling libc v0.2.146
   Compiling rustdemo v0.1.0
(/Users/louisliu/rust_workstation/go-rust-demo/rustdemo)
    Finished release [optimized] target(s) in 2.84s
```

在rust库的 target/aarch64-apple-darwin/release 目录下将会生成一下两个文件：librustdemo.a、librustdemo.dylib，其中.a结尾的为静态链接库，.dylib结尾的为动态链接库。

3.6 提示

按照惯例来说，我们一般会把生成的头文件和链接库文件都拷贝到rust项目根目录下，方便外部调用者直接拷贝使用：

```
Python
➜  rustdemo git:(master) ✗ ls
Cargo.lock        Cargo.toml        librustdemo.a
librustdemo.dylib rustdemo.h        src               target
```

四、golang调用rust

4.1 调用静态链接库

```
Python
package main

/*
   #cgo CFLAGS: -I./rustdemo
   #cgo LDFLAGS: -L${SRCDIR} -lrustdemo
   #include <stdlib.h>
   #include "./rustdemo.h"
*/
import "C"
```

```
import (
	"fmt"
	"unsafe"
)

func main() {
	s := "Go say: Hello Rust."

	input := C.CString(s)
	defer C.free(unsafe.Pointer(input))
	o := C.rustdemo(input)
	output := C.GoString(o)
	fmt.Printf("Go got: %s\n", output)
}
```

在这里我们使用了 `cgo` ，在 `import "C"` 之前的注释内容是一种特殊的语法【中间一定不能插入空行】，这里是正常的 C 代码，其中需要声明使用到的头文件之类的。

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。

CFLAGS 通过 -I./rustdemo 将 rustdemo 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} 将编译后 rustdemo 静态库所在目录加为链接库检索路径，-lrustdemo 表示链接 librustdemo.a 静态库。

4.2 调用动态链接库

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）。

因为动态库和静态库的基础名称都是 librustdemo，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：

```python
package main

/*
   #cgo CFLAGS: -I./rustdemo
   #cgo LDFLAGS: -L${SRCDIR} -lrustdemo
   #include <stdlib.h>
   #include "./rustdemo.h"
*/
import "C"
import (
      "fmt"
      "unsafe"
)

func main() {
      s := "Go say: Hello Rust."

      input := C.CString(s)
      defer C.free(unsafe.Pointer(input))
      o := C.rustdemo(input)
      output := C.GoString(o)
      fmt.Printf("Go got: %s\n", output)
}
```

编译时 GCC 会自动找到 librustdemo.a 或 librustdemo.dylib 进行链接。

无论是使用动态链接库还是静态链接库，最终的运行结果都是一致的：

```python
Rust got: "Go say: Hello Rust."
Go got: Go say: Hello Rust. Rust say: Hello Go
```

五、潜在的坑

5.1 声明周期的问题

golang是一种自带垃圾回收机制的语言，而rust是一种需要自己管理内存的语言，如果处理不好，会出现访问已经被回收的内存的问题。

5.2 线程的问题

golang的线程模型较为特殊，每个线程与系统线程不是一一对应，在进行非 Go 的调用时，会把当前 goroutine 放在一个系统线程上，使用这个系统线程后完成后续的调用流程。如果我们要依赖线程的LTS来进行一些逻辑处理，较为容易出现状态紊乱。