实战：使用 Rust 开发动态链接库并在 Golang中使用（上）| Rust 培养提高计

Databend
已认证的官方帐号

> 本篇文章是受Databend团队邀请，作为《Rust培养提高计划》公开课程的讲稿而书写的。
> 课程相关录像可以在Databend的Bilibili账号【Databend】找到
> 讲师本人的微信公众号是：【极客幼稚园】
> 讲师本人的个人博客是: blog.ideawand.com
> 讲师本人的B站账号是:【爆米花胡了】（注意胡是胡萝卜的胡） space.bilibili.com/5004...

特别提示： 该文长 26000+ 字， 阅读时间可能需要超过 60 分钟。 建义先保存收藏观看。 本次分享只是 Rust FFI 中的一部分，其中目录中标灰的内容还没讲到，还会有后续的更新。 可以关注 Databend 公众号了解最新动态。

课程 Demo 代码： github.com/myrfy001/rus...

Rust 培养提高计划回放 : t.cn/A6M4JIOx

分享背景：

在我所加入的各种 Rust 讨论群中，大家经常问的一个直击灵魂的问题就是：你们在生产环境中用Rust了吗?

的确，现在掌握 Rust 的人还相对较少，很多学习 Rust 的同学可能是出于自己的兴趣爱好，实际并无法在工作中有机会实际使用。我本人也是在两年前接触的 Rust，中间一直断断续续地学习，但大部分停留在看书、看教程、看文档、看别人吐槽的阶段，自己写过的代码也都是不超过几百行的小 Demo。虽然后来在 B 站做过一系列《Rust过程宏实战系列教程》，那也是自己学习Rust的副产品，纸上谈兵，和实际工作没有任何关系。直到三个月前，我才下定决心要在生产环境中实战一把。

本次分享希望可以给大家提供一个思路，通过替换系统中最小模块的方式，逐步用 Rust 来“蚕食”现有系统，实现逐步替换，并逐步积累在生产环境中使用 Rust 的经验。

通过本次分享，我希望可以：

et your hands dirty

- 了解Rust为什么比一些常用的语言（如 Golang、Java 等）更适合开发通用二进制库
- 了解一下 Rust 开发 C-FFI 库的方法，并介绍一些个人实际使用中遇到的问题（也可能是我太菜）
- 了解一下 Golang 中 Cgo 的使用
- 有很多朋友在纠结 Golang 和 Rust 选哪个，通过本次分享，加深对两个语言的了解，为大家做选择多一些参考

分享目录

- Get hands dirty
- Rust 在开发二进制库上的优势
- 为什么选择 Golang 作为调用示例
- 配置一个 Rust 项目，使之能够编译出满足 C-FFI 的动态链接库
- 开发一个 Go 项目，调用 Rust 编写的库
- Case By Case，介绍常见的应用场景
- 字符串的传递
- 结构体以及函数方法
- 回调函数的使用
- FFI 接口处的并发安全问题
- 错误处理
- 性能测试：用 Rust 重写 Go 模块，真的会更快吗?

目标听众

- 对 C 语言有大致了解，大学 C 语言基础就够了
- 熟悉 Go、Python、Java这类具有 GC 的语言，但对 C 这样需要手工管理内存的语言不熟悉的同学
- 了解 Rust，并希望将Rust与其他语言配合使用的同学
- 熟悉 Go 语言，但不了解 cgo 使用方法的同学
- 如果你已经熟练掌握的 Rust、C/C++、Go，那么这个分享可能过于简单，建议直接 Clone Databend 的代码，开始为Databend贡献代码~逃~~

Rust 在开发二进制库上的优势

我们知道，如果两个不同的编程语言希望互相调用对方编写的函数，那么两种语言必须达成很多共识，包括但不限于：

- 各种数据结构在内存中是如何布局存储的
- 函数调用时，参数如何传递，返回值如何传递（例如，是用寄存器来传递，还是用栈来传递？）
- 去哪里申请内存空间（使用的内存分配器是什么？），申请使用的内存空间，何时释放，由谁来释放（有没有GC？）？

如果每一种编程语言都有一套自己分配管理内存、调用函数的规则，那么两种不同编程语言所开发的程序之间如果想进行相互调用，那么难度就会非常大。如果有 N 种编程语言，那么互相调用的可能组合就是 `N * (N-1)`。为了解决这个问题，最好的方法就是找一个公认的标准，大家都来向这个标准靠拢。而出于历史原因，C语言在计算机领域的地位目前是不可撼动的，因此绝大多数编程语言都会尽力提供一些方式，使得自己能够通过C语言的标准来和其他语言互相调用。想对这里深入的同学可以去深入了解一下 `ABI` 和 `FFI` 这两个概念，这里暂不展开。

那么我们来看下一个问题，大家都来兼容C语言定义的标准，事情就这么简单吗？显然不是，毕竟 C 语言已经很老了，如果各种新生语言都拘泥于 C 语言的规范，那怎么能有创新呢?

于是就出现了这样一种局面：由于不同的编程语言在具体实现上千差万别，有各自的特性，大家只在基础的功能上兼容 C 语言，而各种高级特性往往就不能直接满足 C 语言的标准。因此不同的语言对 C 语言所定制的函数调用规范的支持程度也不一样，这就造成了有一些语言之间互相调用很容易，开销也很低，而另外一些语言之间的相互调用会很困难，或者说效率很低。（大家可以回想一下那句名言：没有什么问题是加一层中间件解决不了的，如果有，那就加两层。如果两种编程语言在设计上差异过大，那么势必要在相互调用时进行一些中间的转换工作，而其代价就是互相调用

我们常见的开发语言，通常可以分为两大类：

- A类：是没有虚拟机、运行时，需要开发者自己管理内存的语言，例如 C、C++、Rust 等。
- B类：是具有虚拟机、具有运行时等协助管理内存的组件的语言，例如 Java、Golang、Python、PHP 等。

通常，上述A类语言，都能够完美简洁的支持C语言的调用规范，而 B 类中的语言，通常是部分支持，或者说其对 C 调用规范的支持需要引入额外的开销。上面说了这么多理论的东西，我们来看一个实际的例子：

我们以 Python 和 C 语言交互为例，Python 本身是一种脚本语言，CPython 是 C 语言开发的 Python解释器，接下来的例子我们都以 CPython 为例进行说明。大家如果学习过 Python，可能都会听说 Python 是一门胶水语言，可以非常方便的使用 C 语言开发的库，但是，要知道这层胶水也是有代价的。例如我们想在 Python 中调用一个现成的 C 语言开发的动态库，我们会写下面的代码：

```
from ctypes import cdll, c_char, c_char_p

libc = cdll.LoadLibrary('libc.so.6')
strchr = libc.strchr
strchr.argtypes = [c_char_p, c_char]
strchr.restype = c_char_p

substr = strchr('abcdef', 'd')
if substr:
    print(substr)
```

其中，第 5、6 两行分别指定了 C 编写的函数库中某个函数的入参类型和返回值类型，我们可以想象到，在第8行调用这个函数的时候， `ctypes` 这个库，会执行下列操作：

- 根据第5行指定的入参类型，把两个 Python 字符串对象转换为 C 语言的 `char *` 和 `char` 类型，对于 `char *` 类型，需要确保其指向的字符串是以NULL结尾的。
- 根据转换后的参数调用 C 函数库
- 根据第 6 行定义的返回值类型，将 C 函数返回的一个字节的 `char` 类型结果，重新转换为换一个 `str` 类型的 Python 对象

上面的例子呢，是从 Python 来调用 C，那么反过来，如果我们想从 C来调用 Python 计算一个 1+1 等于几的问题，又会是什么样子的呢？我们知道，Python脚本的运行是需要Python解释器来实现的，因此，哪怕是执行最简单的一个加法计算，我们也要在 C 语言的代码中，先编写初始化 CPython 解释器的代码，初始化一个完整的 CPython 解释器，然后再在这个解释器中运行 Python 的脚本，当然在运行脚本之前，一定还会涉及到把 C 语言的基础数据类型int转换为Python的int类型对象，最后再把存储计算结果的Python对象转换为 C 语言的基本数据类型。

类似的，Golang 在编译时可以通过添加 `-buildmode=c-shared` 来将一个 go 写的项目编译成符合 CFFI 的库文件，但我们可以想一下C语言调用Go编写的库会是什么样子的：Go自己的Runtime（任务调度器、对操作系统 IO 的封装、内存分配器等等）得打包到库里面，否则没法调度协程、没办法分配内存、没办法做 IO，除此之外还有其他很多问题。

而对于Rust开发的库文件呢？由于 Rust 几乎没有 Runtime，内存管理可以和操作系统默认保持一致，系统级编程语言可以直接和系统调用打交道没有额外开销，因此我们可以写出小而美的库文件。

通过上面的例子加深理解以后，我们就可以引出一些结论：

- A类中的语言相互调用，往往是最原生、最高效的。
- B类中的语言相互调用，往往伴随着巨大的额外开销，甚至开销过大以至于失去折腾这些跨语言调用的价值。
- 每种语言都可能处于调用者和被调用者两种角色，而一门语言扮演两种角色时，其支持程度可能是不一样的：

类语言通常比较新，

- A类语言如果想调用B类语言开发的函数，就要看B语言的诚意，因为A类语言是前辈，它们对新的语言特性、实现方式一无所知，后辈们如果希望得到前辈的重用，那就要自己提供出满足前辈价值观的接口。
- 通常，B类调用A类会容易实现一些（只需要做类型转换），而A类调用B类则会困难一些（需要初始化B类语言的运行时、解释器等）。多数语言对B类调A类的支持会优于A类调B类。

那么，回到这一小节的核心问题：Rust 在开发二进制库上的优势是什么？通过上述的分析，如果你希望自己开发的二进制库能够有最好的兼容性，能够被更多的语言来调用，那么最好选择用A类语言来开发，而A类语言中，大部分都是古董级别的语言。Rust是为数不多的【属于A类】的【新一代】开发语言，所以，我很看好它~

为什么选择Golang作为调用示例

- Golang 目前应用非常广泛，熟悉 Golang 的小伙伴们也比较多，这样可能会有更多的受众。
- 最近使用Rust或Go开发的云原生项目增长非常快，因为这两种语言的应用场景在云原生方面有重合，所以研究一下二者的融合就很有意思。
- Golang 具有自己的运行时、自己的栈结构、自己的内存分配器，因此相比较于Python这类胶水语言，Golang调用CFFI函数库更有难度，我们喜欢做一些有挑战的事情。
- 正因为其略微复杂，我们才可以更好的思考一些问题，更好的领会跨语言调用的核心思想。

配置一个Rust项目，使之能够编译出满足C-ABI的动态链接库

由于这是一个视频直播分享的讲稿，为了能够快速给大家进行演示，我事先准备好了各个环节需要使用的代码，大家可以先克隆 `https://github.com/myrfy001/rust_golang_ffi_demo` 这个 GitHub 仓库，并切换到 `example_1` 这个分支上，来查看第一个示例的代码：

这里呢，我们可以看到顶层有 `rust` 和 `golang` 这两个文件夹，里面分别存储了示例中所使用到的Rust工程以及 Golang 工程，我们先看 Rust 文件夹下的内容：

```
.
├── Cargo.lock
├── Cargo.toml
└── src
    ├── ffi.rs
    ├── lib.rs
    └── my_app.rs
```

首先，这个文件夹结构是通过 `cargo new --lib rust` 命令建立的初始架构，然后在src目录下自行新建了 `ffi.rs` 和 `my_app.rs` 两个文件。

先来查看一下 `Cargo.toml` 文件，其中以下的两行是我们自己修改加入的：

```
[lib]
crate-type = ["cdylib"]
```

这里的 `crate-type = ["cdylib"]` 指示我们的这个crate要编译输出一个符合C语言调用规范的动态库文件，注意这里是 `cdylib` ，不是 `dylib` ，如果配置为 `dylib` ，则输出是符合rust调用规范的动态库，只能在rust语言编写的项目之间互相调用，不能跨到其他语言上使用。

再来介绍一下src目录下的结构：

- `my_app.rs` 代表的是我们编写普通Rust代码的地方，我们开发的主要Rust逻辑都写在这个文件中。这只是一个案例，在真实项目中， `my_app.rs` 对应的可能是一个包含数百个源代码文件的庞大的Rust项目。
- `ffi.rs` 是我们的FFI边界，在这个文件中，我们要把Rust提供的各种高级特性适配到经典的C-FFI形式的函数上。
- `lib.rs` 是我们整个库的入口，里面只有两行 `mod` 声明语句，用于引入 `my_app.rs` 和 `ffi.rs`

看到以下代码：

```
pub fn my_app_simple_rust_func_called_from_go(arg1: u8, arg2: u16, arg3: u32) -
  arg1 as usize + arg2 as usize + arg3 as usize
}
```

这段代码非常简单，它是一个普通的 Rust 函数，将 3 个入参相加后返回，因为这是一个最简单的示例，所以入参选择了 3 个不同的基本数据类型。

为了将这个函数通过 C-FFI 暴露给其他程序，我们打开 `ffi.rs` ，可以看到以下内容：

```
use crate::my_app;

#[no_mangle]
fn simple_rust_func_called_from_go(arg1: u8, arg2: u16, arg3: u32) -> usize {
    my_app::my_app_simple_rust_func_called_from_go(arg1, arg2, arg3) as usize
}
```

因为入参和返回值都是基本数据类型，所以这一层包装看起来有点多余，但是先别急，后面的例子就会看到ffi这一层包装的作用了。另外一个很重要的点就是 `#[no_mangle]` 这个属性标签，它告诉Rust编译器不要在输出的符号表中对这个函数的函数名做混淆，因为在跨语言的函数调用中，知道被调用函数的名字是非常重要的，否则链接器就无法在符号表中对应到这个函数的入口地址。默认情况下，Rust 编译器认为我们定义的函数都是由 Rust 程序调用的，这时编译器默认对函数名进行了混淆，为什么要混淆的原因大家可以参考：internals.rust-lang.org... 以及 rust-lang.github.io/rfc...。

虽然外部看起来函数名混淆之后变得人类难以理解，但rust编译器自己心里还是非常清楚这些对应关系的。但是，如果这个函数要被其他语言调用，其他语言在编译链接的时候可不知道rust编译器是怎么混淆的，其他语言的链接器只能拿着函数名去符号表里找，如果混淆了，肯定对不上，因此，在开发ffi接口时，必须告诉编译器，不能对这个函数的名字做混淆。

以上就是我们所编写的第一个 Rust 侧的项目代码，执行 `cargo build` 后，我们会在 `target/debug/` 目录下看到名为 `librust.so` 或 `librust.dylib` 的文件，因为我演示的系统是 Mac OsX 系统，所以我这里得到的是 `librust.dylib` 这个文件。这个文件名的 `lib` 前缀是*nix 操作系统上库文件的固定前缀，而 `rust` 后缀是因为我们在 `Cargo.toml` 中定义的 `name = "rust"` 。

开发一个Go项目，调用Rust编写的库

> 本篇文章虽然是介绍在 Go 中调用 Rust 编写的函数库，但其中以 C-FFI 为界，在 Golang 视角中，其看到的只是一个符合 C-FFI 标准的函数库，本篇文章介绍的与 cgo 有关的知识点，可以应用在其他任何语言所开发的符合 C-FFI 标准的软件库上，不仅限于 Rust 开发的软件库。

还是在代码库 `example_1` 这个分支上，这次我们进入 `golang` 这个目录，可以看到其中有4个文件：

```
.
├── ffi_demo.h
├── go.mod
├── main.go
└── main_test.go
```

其中， `go.mod` 是通过 `go mod init` 创建出来的，无需多说， `main.go` 是我们的核心文件，同时我们在 `main_test.go` 中编写测试用例。这些都是一个Golang项目的常规操作，接下来我们重点看一下 `ffi_demo.h` 这个文件。

在 Golang 中调用符合 C-FFI标准的函数库，我们就要借助 cgo，从名字上就可以看出来我们需要一些和 C 语言有关的东西，这里的 `ffi_demo.h` 是一个C语言的头文件，如果您对 C 语言完全不了解，那么可能需要去自己补充一下相关的知识。不过，为了能够继续看完这个教程，你可以先把

少个参数，每个参数是什么类型的，以及返回值是什么，有了这些东西，编译器就可以按照C-ABI的调用规范生成调用的机器指令了。

下面我们可以看一下这个 `ffi_demo.h` 的内容：

```
#include "stdint.h"

uintptr_t simple_rust_func_called_from_go(uint8_t arg1, uint16_t arg2, uint32_t
```

> 为了简洁明了，这个头文件是我手写的，写的并不是很规范，对于一个大项目，自己手写头文件比较坑，可以使用 `cbindgen` 这个工具来自动生成，有兴趣的同学可以自己去学习一下。

这个文件中，第一行引入了 `stdint.h` 这个文件，这一步主要是为了引入 `uintptr_t` 等等这些类型的定义，rust中的数值类型对应到C语言数据类型的对应关系，大家可以参照cbindgen工具给出的映射表，参见：github.com/eqrion/cbind...

文件的第二行，按照C语言的语法：

- 首先标明函数的返回值类型是一个与处理器位宽一致的数据类型。
- 接下来是函数名，这个函数名必须与rust中添加了 `#[no_mangle]` 标签的函数名一直，因为在链接器链接的过程中，通过符号表查找函数地址的时候，就要靠这个名字
- 后面括号中依次写出了每一个参数的类型，这个类型也一定是和rust中声明的函数签名一致的，只不过它是用C语言的语法来书写的。

然后说一下关于这个 `ffi_demo.h` 文件存放位置的问题，如果是一个正规的开源库项目，这个头文件应该是随着rust项目发布的，各种不同的语言在使用这个rust开发的库时，都可以引用这个头文件，所以它不应该出现在我们golang这个文件夹下面，而是应该放在类似 `/usr/local/include` 这类目录下，但是这为了演示方便，我们把它放在了 `golang` 这个目录里。

关于 `ffi_demo.h` 这个文件，先介绍这些就够了，我们回到我们的主角： `main.go`

```
package main

/*
#cgo CFLAGS: -I.
#cgo LDFLAGS: -L../rust/target/debug -lrust

#include "ffi_demo.h"*/
import "C"


func SimpleRustFuncCalledFromGo() {
  arg1 := 123
  arg2 := 1234
  arg3 := 1234567

  cArg1 := C.uint8_t(arg1)
  cArg2 := C.uint16_t(arg2);
  cArg3 := C.uint32_t(arg3);
  ret := C.simple_rust_func_called_from_go(cArg1, cArg2, cArg3)
  if int(ret) != arg1 + arg2 + arg3 {
    panic("SimpleRustFuncCalledFromGo Error")
  }
}
```

上述第9行 `import "C"` 这一行的出现，标明了这个go文件需要使用cgo，紧接在这一行上面的注释内的内容，会被cgo处理。**注意第9行的 `import "C"` 和它上面的注释之间不能有空行。**

- `#cgo CFLAGS: -I.` 这一行表示告诉C【编译器】，增加一个搜寻头文件的路径，这里的 `-I.` 中的 `.` 表示当前路径，而后面我们在启动测试程序是，就是将 `golang` 这个目录作为当前工作目录，我们的 `ffi_demo.h` 这个文件也就放在工作目录中，这样编译器就能在当前目录下搜索到 `ffi_demo.h` 了。
- `#cgo LDFLAGS: -L../rust/target/debug -lrust` 这一行告诉C【链接器】两件事：
- `-L../rust/target/debug` 表示要增加一个库文件的搜索路径
- `-lrust` 表示要搜索一个叫做 `rust` 的库文件，按照*nix的规范，实际上就是查找一个叫做 `librust.so` 的文件
- `#include "ffi_demo.h"` 这一行就是一段标准的C语言代码，指示C语言编译器引用 `ffi_demo.h` 文件中编写的代码。

> 广告时间：如果大家对*nix系统下动态库的查找方式感兴趣，可以阅读笔者的另一篇文章：《一文读懂Linux下动态链接库版本管理及查找加载方式》

所有在 `import "C"` 这一行上面注释中的C语言代码里定义的函数、结构体等，都会在编译时被认为是在 `C` 这个模块中定义的，因此在这个 `main.go` 文件中接下来的地方，我们就可以用 `C.xxxx` 的形式，来在go语言中访问C语言里面定义的一些内容。

接下来的12~24行定义了一个名为 `SimpleRustFuncCalledFromGo` 的 Go 函数，其中：

- 前三行定义了3个go变量，其类型都是int类型
- 接下来三行，我们从这三个go变量创建了3个调用C库时需要使用的变量，之所以这样做，是因为 `arg1` 、 `arg2` 、 `arg3` 这三个变量在go中默认都是int类型的，也就是说其所占用的内存空间都是与处理器位宽一致的，而我们在 `ffi_demo.h` 中定义的三个参数，分别只占用1、2和4个字节，因此我们需要做一下类型的转换，得到了 `cArg1` 、 `cArg2` 、 `cArg3` 这三个变量，其类型分别是 `C.uint8_t` 、 `C.uint16_t` 、 `C.uint32_t` 。
- 再下面的一行，通过 `C.simple_rust_func_called_from_go` 的形式，表示我们要通过cgo来调用一个外部C库中名为 `simple_rust_func_called_from_go` 的函数：
- 之所以我们能通过 `C` 这个模块来访问 `simple_rust_func_called_from_go` ,是因为我们在代码第7行加载了 `ffi_demo.h` ,而 `simple_rust_func_called_from_go` 定义在 `ffi_demo.h` 中。
- 在编译链接时，链接器需要在多个指定的外部候选库中寻找需要链接的函数，由于我们在第5行告诉链接器要添加一个名为 `librust.so` 的文件作为可能要被链接的候选，而这个库文件的符号表中恰好暴露了一个名为 `simple_rust_func_called_from_go` 的函数，于是链接器就知道当要调用这个函数的时候，要把PC指针跳转到 `librust.so` 这个动态库内部对应的地址上。
- `librust.so` 之所以会暴露 `simple_rust_func_called_from_go` 这个符号，是因为我们在写rust代码的时候添加了 `#[no_mangle]` 标签
- 这个函数调用执行完以后，其返回被存到了ret这个变量中，这个变量的类型是 `C.ulong` ,为了将 `C.ulong` 与go的内置int类型进行比较，我们需要再做一次类型转换。
- 这里可以稍微注意一下，我们在 `ffi_demo.h` 中定义了函数的返回值类型是C语言中的 `uintptr_t` ，这个类型对应到cgo中变为了 `C.ulong` 类型。在跨越FFI边界的时候，类型神马的已经完全不存在了。数据也好，指针也罢，无非就是一串特定的二进制比特序列而已，类型系统告诉编译器如何处理使用这一段比特序列。
- 至于为什么 `ffi_demo.h` 文件中定义的 `uintptr_t` 会变为cgo中的 `C.ulong` ，这个大家也没必要死记硬背，我们可以借助编译器的错误提示信息来获取到底是什么类型。比如以后你遇到了一个不知道是什么类型的东西，那就人为制造一个类型不匹配，编译器会给出响应的提示。例如，我们可以把上面的 `if int(ret) != arg1 + arg2 + arg3 {` 这一行改为 `if ret != arg1 + arg2 + arg3 {` ,编译器就会提示无法将 `C._Ctype_ulong` 和 `int` 做比较，这时候我们就知道ret对应的类型实际上是 `C.ulong` 了

以上，我们捋顺了go调用简单C函数的流程

看到这里，你可能会说，何必多此一举呢？我在go里面，直接把 `arg1` 、 `arg2` 、 `arg3` 这三个参数定义成 `uint8` 、 `uint16` 和 `uint32` 不就行了嘛。但这样也是不行的：

> **通过cgo调用c函数时，参数都要使用 `C` 这个模块内部提供的类型。而返回值也一定是 `C` 模块内定义的某种类型**

`C.uchar` 、 `C.short` 、 `C.int` 这些类型，都是cgo内置的类型。之所以要有这样的限制，大家可以回忆一下本文开头在介绍跨语言调用时、提到过为了减少不同语言之间调用的组合爆炸问题，

大家都公认把C语言的调用规范作为跨语言调用的标准，那么，我们在跨语言调用的时候，就只能用C语言知道的那些东西。

`C` 这个虚拟的模块所起到的作用，其中一个就是从类型系统的层面上，限制你能使用的类型。例如C语言根本不知道Golang里面的Channel和闭包是什么东西，因此你就不可能在 `C` 这个模块中找到和Channel、闭包等有关的东西。从这一点，也希望大家可以慢慢体会“跨语言调用可能会有一些性能开销”这件事，当然，这个例子里面遇到的转换应该是最简单廉价的一种转换，后面我们会看到更复杂、开销更大的转换。

现在，我们运行 `main_test.go` 里面的测试用例，应该可以看到测试用例顺利执行。

字符串的传递

本小节将提升一些难度，为大家介绍跨语言传递字符串的方法。之所以难度会有提升，是因为相比上一关所有变量都是栈分配而言，本小节的字符串类型涉及到了堆内存的使用。而在使用堆内存的时候，必须处理好谁申请，谁释放的问题；除此之外，另一个头疼的问题就是C语言中的字符串规定以Null结尾，字符串本身就是一个指针，不包含长度信息，而Rust中的String也好，&str也好，都使用了额外的空间保存字符串长度，而真正的字符串在内存中不能保证是以Null结尾的，这就会引入额外的转换操作。

> 预防针
> 在接下来的内容中，会遇到一些非常规的做法，这里之所以要介绍这些非常规的做法，目的不是为了鼓励大家在代码中去这么使用，而是给大家展示各种各样的情况，让大家了解到无论上层接口再怎么变，使用方法再怎么稀奇古怪，其底层核心不变的就是内存的分配与释放。借助unsafe rust，大家可以获得像C/C++一样对内存的完全掌控能力。实际使用中FFI边界上的情况非常灵活多变，但只要内存管理能搞清楚，其他的事情都不是大问题。

了解库函数的核心功能逻辑

这一部分的代码我们切换到项目的 `example_2` 分支，然后打开 `rust/src/my_app.rs` 文件，可以看到我们定义了以下四个新的函数：

```
pub fn my_app_receive_string_and_return_string(s: String) -> String {}

pub fn my_app_receive_str_and_return_string(s: &str) -> String {}

pub fn my_app_receive_str_and_return_str(s: &str) -> &str {}

pub unsafe fn my_app_receive_string_and_return_str<'a>(s: String) -> (&'a str,
```

这四个函数分别列举了输入参数和返回值取 `String` 和 `&str` 两种类型时所有的排列组合情况，而它们的功能都是一致的：

- 当输入字符串的长度小于15个byte的时候，返回完整的字符串，而超过15个byte的时候返回前15个byte

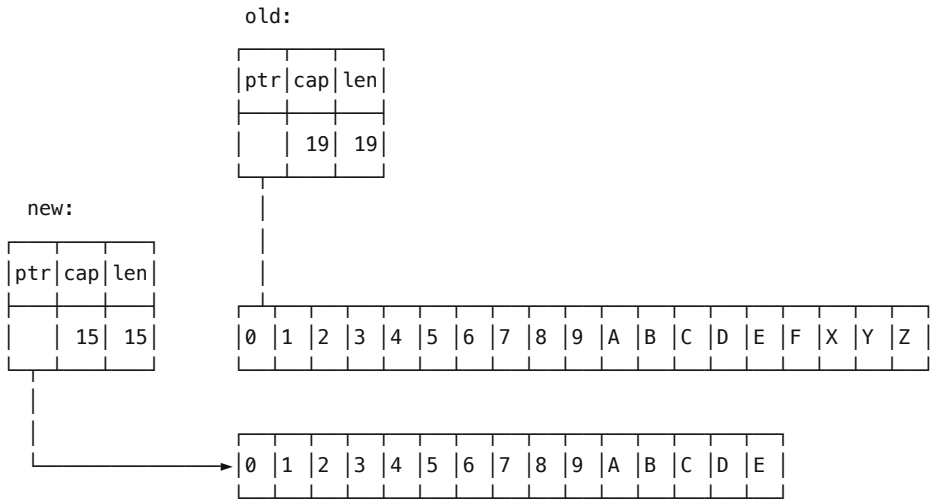
通过对 `String` 和 `&str` 的排列组合，我们要强化大家对Rust中字符串相关的内存分配情况的理解，知道在对字符串做处理时什么时候会发生内存分配和拷贝，我们依次来看：

接收String，返回String

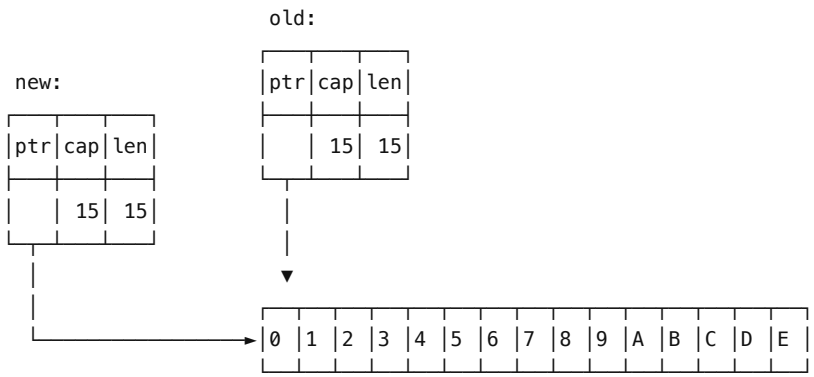
```
pub fn my_app_receive_string_and_return_string(s: String) -> String {
  if s.len() > 15 {
    // this path has new memory alloc on heap
    s[0..15].to_string()
  } else {
    // this path doesn't have new memory alloc on heap
    s
```

```
    }
}
```

上述代码，根据字符串长度不同，可能发生一次堆内存分配，也可能不发生堆内存分配。如果发生了堆内存分配，则可以用下图来表示，长度为19的字符串，经过截断后， `to_string()` 调用会把前15个字节复制出来，这时发生了一次堆内存分配，函数返回后，长度为19的字符串的字符串头（栈内存）和字符串内容(堆内存）都被释放，返回的是新的字符串头，指向新的堆内存。



如果没有发生堆内存分配，则函数返回前后的情况如下：堆内存被复用，入参对应的字符串头（栈内存）被释放，而返回一个新的字符串头（栈内存）。

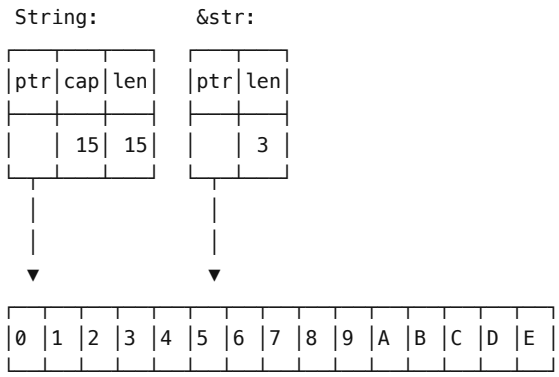


> 出于篇幅的限制，后面不能每个case都画图来说明。在继续后面的介绍之前，大家一定要理解这个字符串在内存中的表示形式，并且在看到后面的代码时，能想想出对应的内存布局。

接收&str，返回String

```
pub fn my_app_receive_str_and_return_string(s: &str) -> String {
  // both path alloc new memory
  if s.len() > 15 {
    s[0..15].to_string()
  } else {
    s.to_string()
  }
}
```

上述代码中两条路径都有涉及到内存分配。画个图帮大家回忆一下 `&str` 和 `String` 的区别，底层的堆内存由 `String` 持有， `&str` 是一个胖指针，只能临时指向其中的某段地址，因此，图中 `&str` 对应变量的生命周期不能超过 `String` 的存活时间。



接收&str，返回&str

下面两条路径都没有涉及到资源分配

```
pub fn my_app_receive_str_and_return_str(s: &str) -> &str {
  // neither path alloc new memory
  if s.len() > 15 {
    &s[0..15]
  } else {
    s
  }
}
```

接收String，返回&str

这个函数通过Unsafe实现了安全Rust所绝对不允许的事情，即凭空返回一个堆内存的引用。再次提醒，这只是一个示例，为了帮助大家加深对Rust手动内存管理的理解，自己写代码的时候不要写这种容易被打的代码。这个函数两条路径也都没有进行堆内存分配。

```
pub unsafe fn my_app_receive_string_and_return_str<'a>(s: String) -> (&'a str,
  // this function is only used as an example to show that we can use unsafe
  // rust to turn an owned type to a reference, you should not write such code
  // in production code. It's a very ugly api design.


  // neither path alloc new memory
  let my_slice = if s.len() > 15 {
    &*(&s[0..15] as &str as *const str)
  } else {
    &*(&s as &str as *const str)
  };

  // you can replace the following two lines using s.into_raw_parts()
  // s.into_raw_parts() internally use ManuallyDrop too
  // I use ManuallyDrop explicit here to show you how memory is managed
  // The reason why we need to return the ptr, len and cap is that we need ther
  // to rebuild the String header, we need to rebuild the string header to
  // free memory.
  let s = ManuallyDrop::new(s);
  (my_slice, s.as_ptr(), s.len(), s.capacity())
}
```

安全的Rust之所以不允许在函数中返回一个Srting的引用，是因为在函数返回时，String会被drop，对应的堆内存被回收，导致引用到被回收的堆内存空间，而上面的代码中，通过ManuallyDrop的包装，我们使得String不会在函数返回时被drop，这样String所持有的堆内存就被“遗忘”了，于是我们就可以放心的引用这块堆内存了。之所以“遗忘”要打引号，是因为我们不能真

这块内存，而这个线

索就是函数返回值的第2、3、4个参数，通过指针、容量、实际占用长度这三个参数，我们就可以重新在栈内存上构建出String头部的结构体，从而将"遗忘"的String找回来。

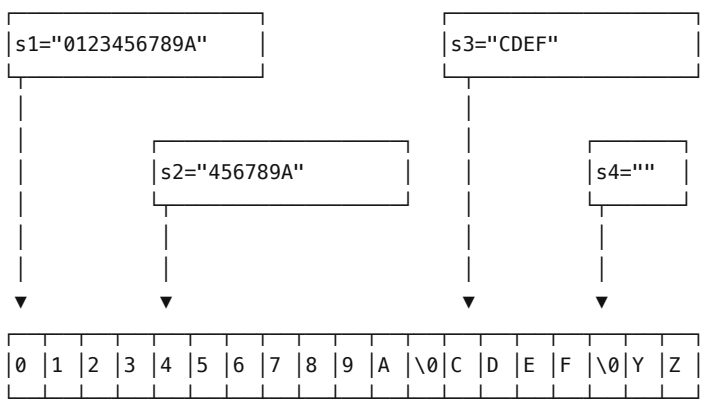
到此为止，我们完成了对 `my_app.rs` 的分析，也就是了解了我们所编写的库要做的本质功能是什么，接下来，我们开始包装FFI接口。

包装字符串传递的FFI接口

打开 `rust/src/ffi.rs` ，我们可以看到新增了几个接口函数的定义，其中每一个函数我都写了详细的英文注释在里面，大家可以选择直接看代码中的注释，或者对照我的博客阅读中文的讲解，新增的函数如下：

```
#[no_mangle]
pub fn receive_str_and_return_string(s: *const c_char) -> *const c_char{}

#[no_mangle]
pub fn receive_string_and_return_string(s: *const c_char) -> *const c_char{}

#[no_mangle]
pub fn receive_str_and_return_str(s: *const c_char) -> *const c_char{}

#[no_mangle]
pub fn receive_string_and_return_str(s: *const c_char, new_ptr: *mut *const c_(

#[no_mangle]
pub unsafe fn free_string_alloc_by_rust_by_raw_parts(s: *mut c_char, len: usize

#[no_mangle]
pub unsafe fn free_cstring_alloc_by_rust(s: *mut c_char){}

#[no_mangle]
pub fn receive_str_and_return_str_no_copy(s: *const c_char, new_ptr: *mut *cons
```

我们先来浏览一下前三个接口函数的定义，可以发现，虽然在 `my_app.rs` 中，我们定义的字符串参数和返回值有 `String` 和 `&str` 的各种组合，但到了FFI接口上，统统变成了 `*const c_char` ，这就是因为C语言中的字符串定义很简单：一个以Null结尾的字节数组，如下所示。



对比一下Rust的 `String` 、 `&str` 和C语言中 `char*` 的区别：

- `String` 头占3个字长， `&str` 胖指针占2个字长， `char*` 指针占1个字长
- `String` 和 `&str` 里面都有长度记录，可以O(1)复杂度获取长度，而 `char*` 要从头开始数，是O(N)复杂度
- `String` 和 `&str` 所指向的字节序列一定是满足UTF-8编码的字符串（Rust规范），而 `char*` 里面存储的可以是任何字符编码（UTF-8、GBK、ASCII等等），甚至可以是非字符编码，比如

- `String` 和 `&str` 可以在中间出现Null字节，但是 `char*` 中间不可以，一旦出现就代表着字符串到这里结束了。

可以看到，Rust和C规范中对于字符串的定义有着很大的差异，而Rust看中确定性，要把一切不确定因素消灭在萌芽阶段，因此，在FFI边界上，我们需要通过一系列手段来保证C语言世界的 `char*` 可以安全的转换为Rust认可的字符串，为了实现这个目标，Rust为我们提供了两个数据类型： `CString` 和 `CStr`：

- `CString` 对应的是 `String` ，用来表示具有堆内存所有权的字符串，但内容是满足C字符串标准的，也就是说内容不一定是UTF-8编码的，且一定没有Null。
- `CString` 通常用来在Rust一侧分配堆内存后，把Rust生成的字符串传递给外部系统使用，堆内存所有权在Rust手里，后续要由Rust来释放。
- `CStr` 对应的就是 `str` ，他同样是一个DST类型，因此通常以 `&CStr` 的形式出现，表示对一段内存地址的引用，其内部存储的数据也不一定是UTF-8编码的，也一定在中间没有Null。
- `CStr` 通常表示外部传入的字符串，字符串对应的内存空间所有权在外部系统，Rust的代码只能临时借用，而不是拥有。

有了这些理论储备以后，我们可以开始看第一个FFI封装函数了：

receive_str_and_return_string:

```
#[no_mangle]
pub fn receive_str_and_return_string(s: *const c_char) -> *const c_char {
    let cstr = {
        // 永远要记得先检查传入的指针是不是空指针
        assert!(!s.is_null());
        // 下面这一行内部实现会遍历s，寻找第一个Null出现的位置，说白了就是做计数，数出来字符
        // &CStr 相比于 *const c_char而言，内部直接记录了长度信息
        // 由于我们要读裸指针，所以需要一个unsafe块
        unsafe{CStr::from_ptr(s)}
    };

    // 下面这一行会再一次遍历底层的字节序列来检查一下这个字节序列是否满足UTF-8编码，如果检查
    // 函数会返回一个&str类型的指针，同样指向原来的地址。这一步相当于做了一个认证，把`&CStr
    // 就可以告诉类型系统：这块内存满足我们Rust对字符串的要求，可以在Safe Rust代码里放心使用
    // 这个操作会复用底层的字节数组，不会发生内存分配和数据拷贝。
    // 由于`&str`是一个引用，它引用的是外部系统（例如Golang）所持有的内存，所以，外部系统必
    // 块内存不会在使用中被释放
    let rstr = cstr.to_str().expect("not valid utf-8 string");

    // 上面的两行代码没有做内存分配，但是却从头到尾遍历了两次字符串。如果这个FFI接口在你的Ho
    // 你需要认真考虑一下性能开销的影响。

    // 现在，我们就可以在Safe Rust的世界来使用外部的字符串了
    let ret = my_app::my_app_receive_str_and_return_string(rstr);

    // 现在，我们得到了一个`String`，这也就意味着上面的函数调用过程中发生了Rust这一侧的内存
    // 我们将要返回给调用者一个指针，但返回的数据必须遵循C的字符串规范才能够穿越FFI的边界，也
    // 要返回一个裸指针，裸指针本身没有字符串的长度信息，并且指向一块以Null结尾的内存区域。
    // 而我们知道，Rust的String不一定能保证字符串结尾的后面一定有一个Null存在。
    // 这时我们就要借助`CString`这个数据类型了，不幸的是，这里又会引入额外的开销：
    // * 首先，它会再次检查整个底层字节数组是否在中间包含Null，因此它会再次遍历字符串。
    // * 其次，它将尝试将 Null字节追加到底层数组的后面，这一步是否会有额外开销取决于底层的缓
    //    是否还有空闲空间，也就是String的len是否小于cap。如果有空间，那么追加一个Null字节，
    //    没有什么开销，但如果没有空间，那么就要进行扩容操作，这会导致一次内存分配以及一次数据
    let c_ret = CString::new(ret).expect("null byte in the middle");

    // 终于到了这里，我们可以通过into_raw()的方法返回一个指向堆内存的指针了。这个方法会消耗
    // c_ret，让编译器暂时“忘掉”这块堆内存。
    // 但是，这块内存迟早要有人来释放，应该怎么释放呢？我们后面再介绍。
    c_ret.into_raw()
```

```
    // 看完上面的代码，你必须要掌握的一个核心点：
    // * 这个ffi包装函数的入参所对应的内存是由调用者拥有的（比如Golang），因此也必须由外部
    //   来决定何时释放
    // * 返回值是由Rust申请的，那么将来也一定是由Rust来执行释放
}
```

大家看完上面这一顿操作以后，可能会对FFI接口的效率失去信心……我就是想传递一个字符串，怎么有这么多额外的开销啊……。

对于这个问题，大家先不要慌，上面给大家展示的流程可以说是最安全的一个流程，这些额外开销换来的是更高的安全性，为了优化，我们有几个方向可以尝试：

- 通过人为的约定，减少一些不必要的检查
- 通过调整API接口的设计，显式传递字符串长度
- 通过调整API接口设计，将频繁的调用转化为少量批次调用，以减小开销
- 避免在Hot Path上使用转换开销大的接口
  关于提升性能，我们会在文章后后面进一步讨论，通过做性能测试的方法，来对比不同方案的性能提升。

接下来还有两个函数，我们也依次来看一下他们在内存使用上有什么差异。为了缩短篇幅，相同的注释内容就被删除了。

receive_string_and_return_string:

```
#[no_mangle]
pub fn receive_string_and_return_string(s: *const c_char) -> *const c_char {
    let cstr = {
        assert!(!s.is_null());
        unsafe{CStr::from_ptr(s)}
    };

    // to_str() 会遍历一次String来检查是否满足UTF-8，to_string() 会分配一次内存并且做一
    let r_string = cstr.to_str().expect("not valid utf-8 string").to_string();

    // 下面这一行调用，根据传入的字符串长度，可能进行内存分配，也可能不进行分配。
    let ret = my_app::my_app_receive_string_and_return_string(r_string);

    let c_ret = CString::new(ret).expect("null byte in the middle");
    c_ret.into_raw()

    // 重要提示：
    // 和上一个函数一样，输入数据的内存是被Golang持有的，返回值是被Rust持有的
    // 返回的指针指向c_ret申请的的堆内存，该段内存因为into_raw()方法，暂时被编译器忘掉，函
    // r_string 和 ret 都是临时变量，最终返回的字符指针指向的堆内存地址是c_ret变量持有的，
    // r_string 和 ret 所分配的堆内存都会在函数返回时被回收。
}
```

receive_str_and_return_str

```
#[no_mangle]
pub fn receive_str_and_return_str(s: *const c_char) -> *const c_char {
    let cstr = {
        assert!(!s.is_null());
        unsafe{CStr::from_ptr(s)}
    };

    let rstr = cstr.to_str().expect("not valid utf-8 string");

    // 下面这个函数调用前面会进行内存分配，Rust负责申请一块底层的字符缓冲区，
```

```
    let ret = my_app::my_app_receive_str_and_return_str(rstr);

    // 但是，问题来了，如果上面的函数调用返回的是一个长字符串的子切片，那么这个子切片就不会是
    // 因为&str是一个Rust概念里的胖指针，里面存了长度，所以这对Rust不是个问题。但在FFI的边
    // 使用C语言规范来交流，所以这就成了一个大问题。因此，我们还是需要再创建一个CString，于
    // 内存分配和拷贝。
    let c_ret = CString::new(ret).expect("null byte in the middle");
    c_ret.into_raw()

    // 重要提示：
    // 这是一个用来演示Rust在FFI边界上处理字符串引用所引入的开销的例子。在纯Rust中，
    // `my_app::my_app_receive_str_and_return_str(str)`这个函数的参数和返回值都是引
    // 类型，所以我们可以避免数据的复制。
    // 但是在FFI的边界上，根据FFI包装函数实现方式的不同，返回值可能可以复用内存，避免拷贝，
    // 也可能需要重新分配内存并拷贝。
    // 在我们当前的这个例子的实现中，我们就无法避免拷贝。如果你想避免拷贝，那么就要重新设计FF
    // 接口的API样式。（我们会在后面的例子中给出例子）
    // 作为这个函数库的作者，你有责任编写一个清晰的使用手册来告诉用户，输入数据与输出数据的内
    // 是怎样被使用的。
    // 最后，和上一个函数一样，输入数据的内存是被Golang持有的，返回值是被Rust持有的
    // 返回的指针指向c_ret申请的的堆内存，该段内存因为into_raw()方法，暂时被编译器忘掉，函
}
```

receive_string_and_return_str

！！！预警！！！这个API的设计非常丑陋，这里只是一个实例，千万不要在生产环境中写这样的代码！

这个接口中使用了二阶指针作为参数，这样做的目的是通过这些参数实现返回多个结果。在C语言的调用规范中，

是不允许一个函数有多个返回值的，为了返回多个结果，我们有两种方式：

- 定义一个结构体来保存多个返回值的内容，然后返回指向这个结构体的指针
- 通过传入指针来修改调用者的内存数据，从而将要返回的值写入到调用者给定的变量中
  这里我们的二阶指针就是使用了第二种方法。

```
#[no_mangle]
pub fn receive_string_and_return_str(s: *const c_char, new_ptr: *mut *const c_
    let cstr = {
        assert!(!s.is_null());
        unsafe{CStr::from_ptr(s)}
    };

    let r_string = cstr.to_str().expect("not valid utf-8 string").to_string();


    let (ret, t_c_origin_ptr, t_len, t_cap) = unsafe{my_app::my_app_receive_st


    let c_ret = CString::new(ret).expect("null byte in the middle");
    unsafe {
        *new_ptr = c_ret.into_raw();
        *c_origin_ptr = t_c_origin_ptr as *const i8;
        *len = t_len;
        *cap = t_cap;
    }

    // 重要提示：
```

持有的
暂时被编译器忘掉，函

```
    // ret是一个临时变量
    // ret 和 t_c_origin_ptr 都指向r_string申请到的堆内存
    // 由于在`my_app::my_app_receive_string_and_return_str()`这个函数内部，我们也让
    // 的堆内存，所以在函数返回时，r_string申请的内存不会被释放，c_origin_ptr所指向的内存
}
```

内存释放的相关函数

在介绍前面几个函数的时候，我们都遗留了一个小问题，就是返回给外部系统的字符串，都是通过CString在堆内存分配后，通过调用 `into_raw()` 让编译器不要自动回收内存，因此为了防止内存泄漏，那就必须要提供一个手段，让调用者可以告诉Rust什么时候可以回收这段内存。因此我们提供了下面两个方法。这两个方法的核心工作都是根据之前留下的线索，重建一个Rust对象来引用之前被"遗忘"的堆内存，之后随着函数调用的返回，这个重建的对象在被销毁的过程中就会顺带释放之前申请的堆内存

```
#[no_mangle]
pub unsafe fn free_string_alloc_by_rust_by_raw_parts(s: *mut c_char, len: usize
  String::from_raw_parts(s as *mut u8, len, cap);
}

#[no_mangle]
pub unsafe fn free_cstring_alloc_by_rust(s: *mut c_char) {
  // 这个方法会再次遍历指针所指向的内存，直到找到Null，从而计算出字符串的长度
  CString::from_raw(s);
}
```

一个零拷贝的FFI接口示例

前面介绍的4个字符串传递，都涉及到了大量的内存分配和数据拷贝，而其中 `my_app::my_app_receive_str_and_return_str()` 这个函数本身设计的意图是避免数据拷贝的，为了在FFI接口上实现这个功能，我们需要定义一套新的API来传递字符串，其核心原理很简单，就是要增加一个返回值来表示字符串的长度，这样就避免了通过遍历Null来计算长度，也避免了截断字符串尾部没有Null的问题。代码如下：

```
#[no_mangle]
pub fn receive_str_and_return_str_no_copy(s: *const c_char, new_ptr: *mut *cons
    let cstr = {
        assert!(!s.is_null());
        unsafe{CStr::from_ptr(s)}
    };

    let rstr = cstr.to_str().expect("not valid utf-8 string");
    let ret = my_app::my_app_receive_str_and_return_str(rstr);
    let c_ret = ret.as_ptr();

    unsafe {
        *new_ptr = c_ret as *const i8;
        *len = ret.len();
    }

    // 重要提示：
    // 在这段代码中没有出现任何`to_owned()` 或 `to_string()`，因此这个API接口不会申请任
    // 我们显示返回了字符串的长度，这样就解决了截取字符串末尾没有Null的问题。
    // 返回的指针所指向的内存，是外部系统分配的内存空间，不是Rust分配的！！！
    // 如果你设计了一个这样的API接口，那么你应该在说明文档中清楚地写出来，返回的内存指针是指
}
```

课后作业：这里我们只是把返回数据的长度进行了额外的存储，但是传入的字符串仍然需要遍历才可以获得其长度。请尝试修改API接口，从而避免对输入字符串的遍历。

在Golang中调用

首先我们要更新一下头文件，打开 `golang/ffi_demo.h` ，我们会发现增加了对应的rust暴露出来的函数定义，有一些函数是我们会用到，暂时大家不用理会，新的头文件如下：

```
#include "stdint.h"

uintptr_t simple_rust_func_called_from_go(uint8_t arg1, uint16_t arg2, uint32_t

char* receive_str_and_return_string(char*);
char* receive_string_and_return_string(char*);
char* receive_str_and_return_str(char*);
// the follow line is a very ugly api design, only used as example, never use t
char* receive_string_and_return_str(char*, char**, char**, uintptr_t*, uintptr_

void free_string_alloc_by_rust_by_raw_parts(char*, uintptr_t, uintptr_t);
void free_cstring_alloc_by_rust(char*);

void receive_str_and_return_str_no_copy(char*, char**, uintptr_t*);
```

可以看到，对于rust中 `*const c_char` 这样的原始指针类型，映射到C语言中变为了 `char*` 类型，其他的大家看一看就好，没有什么复杂的地方。

接下来我们打开 `golang/main.go` 文件，第一处修改是在 `import "C"` 的上方增加了 `#include "stdlib.h"` 这一行。之所以要引入 `stdlib.h` ，是为了后续可以调用 `C.free()` 来执行内存释放，后面我们会详细介绍。

接下来看一下 `PassStringBySinglePointer()` 这个函数，这个函数中定义了一个匿名函数来执行核心的调用逻辑，我们来看一下这个核心匿名函数：

```
  testProc := func(f int, x, y string) {
    goStr := x

    // Memory Alloc in OS allocator And String Copy
    // cStr is not managed by go GC
    cStr := C.CString(goStr)
    defer C.free(unsafe.Pointer(cStr))

    var cStrRet *C.char
    switch f{
    case 1:
      cStrRet = C.receive_str_and_return_string(cStr)
    case 2:
      cStrRet = C.receive_string_and_return_string(cStr)
    case 3:
      cStrRet = C.receive_str_and_return_str(cStr)
    }

    // Memory Alloc in Go runtime And String Copy
    // goStrRet is managed by go GC
    goStrRet := C.GoString(cStrRet)
    C.free_cstring_alloc_by_rust(cStrRet)

    if goStrRet != y {
      panic(fmt.Sprintf("Error, expected %s, got %s", y, goStrRet))
    }
```

上面代码有几个要点：

- 由于函数签名相同，我们用这一个函数来测试三个Rust暴露出来的函数，通过第一个参数来选择调用哪个rust函数。
- `C.CString()` 和 `C.GoString()` 两个内置函数分别用于实现C和Go字符串的相互转换，这里的原理和Rust的原理类似，就不再展开讲了，需要注意的是，这两个函数调用都会导致内存分配和内存拷贝。
- 因为Go是有垃圾回收的，GoString所对应的内存地址是在Go Runtime的内存分配器管辖范围里的，而cgo中为了实现CFFI的调用，需要使用操作系统的内存分配器，Go语言中这两个内存分配器对内存管理的方式不一样，其内存地址区间也不一样，因此需要内存分配与拷贝。
- `C.free()` 调用的是操作系统默认内存分配器的free方法，用来释放掉Go语言通过操作系统内存分配器分配的内存。这部分内存不受Go语言垃圾回收的管理，因此需要手动释放，否则就会内存泄漏。
- Rust函数调用完成后，Rust返回了一个指向Rust分配的内存的指针，这部分内存必须由Rust来释放，因此需要再调用一下Rust库暴露出来的释放接口。

最后，我们就可以通过下面的调用来验证效果了,可以看到，如果超过15个字节的部分，会被截断，而小于15个字节的话，会被原封不动的返回：

```
testProc(1, "极客幼稚园是一个不错的微信公众号", "极客幼稚园")
testProc(1, "Datafuse Lab", "Datafuse Lab")

testProc(2, "极客幼稚园是一个不错的微信公众号", "极客幼稚园")
testProc(2, "Datafuse Lab", "Datafuse Lab")

testProc(3, "极客幼稚园是一个不错的微信公众号", "极客幼稚园")
testProc(3, "Datafuse Lab", "Datafuse Lab")
```

接下来，我们要看一下另一种函数签名的接口API如何使用，代码如下：

```
testProc := func(x, y string) {
  goStr := x
  cStr := C.CString(goStr)  // Memory Alloc And String Copy
  defer C.free(unsafe.Pointer(cStr))

  var cStrRet *C.char
  var cRawStr *C.char
  retCap := C.ulong(0)
  retLen := C.ulong(0)

  C.receive_string_and_return_str(cStr, &cStrRet, &cRawStr, &retLen, &retCap)


  goStrRet := C.GoString(cStrRet)  // Memory Alloc And String Copy
  C.free_string_alloc_by_rust_by_raw_parts(cStrRet, retLen, retCap)

  if goStrRet != y {
    panic(fmt.Sprintf("Error, expected %s, got %s", y, goStrRet))
  }
}
```

可以看到，在这个版本的调用中，我们首先在Go这一侧分配了 `cStrRet` 、 `cRawStr` 、 `retCap` 、 `retLen` 这四个变量，并通过指针的方式将其传递给Rust，Rust通过指针可以直接修改在Go中分配的这些内存中数据的数据，相当于实现了多返回值的效果。

最后，我们来看一下号称零拷贝的Rust FFI接口是如何使用的，代码如下：

```
testProc := func(x, y string) {
```

```
    defer C.free(unsafe.Pointer(cStr))

    var cStrRet *C.char
    retLen := C.ulong(0)

    C.receive_str_and_return_str_no_copy(cStr, &cStrRet, &retLen)

    goStrRet := C.GoString(cStrRet)  // Memory Alloc And String Copy
    // 注意：不同于前面的Go代码，我们在创建完GoString以后，不能在这里调用Rust提供的内存释放
    // 因为cStrRet所指向的内存就是上面cStr所分配的内存，这段内存会被上面的defer语句在函数返
    // 如果我们在这里释放，就会导致内存的二次释放问题。

    // 因为是复用的同一块内存，所以重构出来的Go字符串会和输入的字符串一模一样。因为我们并不能
    if goStrRet != x {
      panic(fmt.Sprintf("Error, expected %s, got %s", x, goStrRet))
    }

    // 但是，通过Rust函数返回的长度信息，我们截取出来我们要的数据
    goStrRetWithLengthLimit := goStrRet[:retLen]
    if goStrRetWithLengthLimit != y {
      panic(fmt.Sprintf("Error, expected %s, got %s", y, goStrRetWithLengthLim
    }

    // 我们的演示案例中，由于没有修改字符串的起始位置，只是调整了它的结尾，所以API返回新的字
    // 当然也可以打到效果，毕竟我们只需要知道一个长度而已。但是如果这个函数的功能是从字符串的
    // 始指针就很重要了。
    if goStr[:retLen] != y {
      panic(fmt.Sprintf("Error, expected %s, got %s", y, goStrRetWithLengthLim
    }
  }
```

可以看到，虽然这几个测试，输入一样，返回值也一样，但其内部的实现方式、对内存的使用方式，有着极大的差异，因此，讲到这里，大家或许会觉得我说的比较啰嗦，但是能够彻底理解他们之间的差异，是进行FFI开发必备的能力。如果大家还有不是很清楚的地方，一定要搞明白。

虽然上面给大家展示了几种不同的内存使用以及参数传递的方式，但是这并不是所有的排列组合，也不一定是最优雅的API接口设计。作为课后作业，大家可以思考一下，传递字符串的API接口还可以怎么设计，能够更加高效、更加简洁?

- 本文作者： Myrfy
- 本文链接： blog.ideawand.com/2021/...

发布于 2 小时前

Go 语言　Rust（编程语言）　编程语言

推荐阅读

Rust“与众不同”特点的汇总

参考《rust程序设计语言》，一点学习笔记分享所有权Rust提供所有权系统，可以让Rust程序无需垃圾回收也可以保障内存安全。所有权

这次只学一点 Rust 语法大概不会怀孕了吧(1)

大家好，这次我来说一说这门名叫Rust的语言的语法。推荐有一点点Rust基础的读者来读，其实零基础

「Rust每日新闻」本周精选 • 第十五期

2 条评论 切换为时间排序

写下你的评论...

一木四水 1 小时前

dylib谁说是rust的库了？那rlib算啥?

赞

Databend (作者) 回复 **一木四水** 47 分钟前

很高兴收到你的回复， 这是一个系例公开课，这次还没讲到 rlib 。 后面可能会有分享。 感谢你也是这方面的高手，也可以加一下我的微信： 82565387

赞 推荐 删除