



下载APP



31 | FFI : Rust 如何和你的语言架起沟通桥梁？

2021-11-08 陈天

《陈天 · Rust 编程第一课》

课程介绍 >



讲述：陈天

时长 18:58 大小 17.37M



你好，我是陈天。

FFI (Foreign Function Interface) ，也就是外部函数接口，或者说语言交互接口，对于大部分开发者来说，是一个神秘的存在，平时可能几乎不会接触到它，更别说撰写 FFI 代码了。

其实你用的语言生态有很大一部分是由 FFI 构建的。比如你在 Python 下使用着 NumPy 愉快地做着数值计算，殊不知 NumPy 的底层细节都是由 C 构建的；当你用 Rust 时，开心地使用着 OpenSSL 为你的 HTTP 服务保驾护航，其实底下也是 C 在处理着一切算法。



我们现在所处的软件世界，几乎所有的编程语言都在和 C 打造出来的生态系统打交道，所以，**一门语言，如果能跟 C ABI (Application Binary Interface) 处理好关系，那么就几乎可以和任何语言互通。**

当然，对于大部分其他语言的使用者来说，不知道如何和 C 互通也无所谓，因为开源世界里总有“前辈”们替我们铺好路让我们前进；但对于 Rust 语言的使用者来说，在别人铺好的路上前进之余，偶尔，我们自己也需要为自己、为别人铺一铺路。谁让 Rust 是一门系统级别的语言呢。所谓，能力越大，责任越大嘛。

也正因为此，当大部分语言都还在吸血 C 的生态时，Rust 在大大方方地极尽所能反哺生态。比如 cloudflare 和百度的 [mesalink](#) 就分别把纯 Rust 的 HTTP/3 实现 quiche 和 TLS 实现 Rustls，引入到 C/C++ 的生态里，让 C/C++ 的生态更美好、更安全。

所以现在，除了用 C/C++ 做底层外，越来越多的库会先用 Rust 实现，再构建出对应 Python ([pyo3](#))、JavaScript (wasm)、Node.js ([neon](#))、Swift ([uniffi](#))、Kotlin (uniffi) 等实现。

所以学习 Rust 有一个好处就是，学着学着，你会发现，不但能造一大堆轮子给自己用，还能造一大堆轮子给其它语言用，并且 Rust 的生态还很支持和鼓励你造轮子给其它语言用。于是乎，Java 的理想“一次撰写，到处使用”，**在 Rust 这里成了“一次撰写，到处调用”。**

好，聊了这么多，你是不是已经非常好奇 Rust FFI 能力到底如何？其实之前我们见识过冰山一角，在[第 6 讲](#) get hands dirty 做的那个 SQL 查询工具，我们实现了 Python 和 Node.js 的绑定。今天，就来更广泛地学习一下 Rust 如何跟你的语言架构起沟通的桥梁。

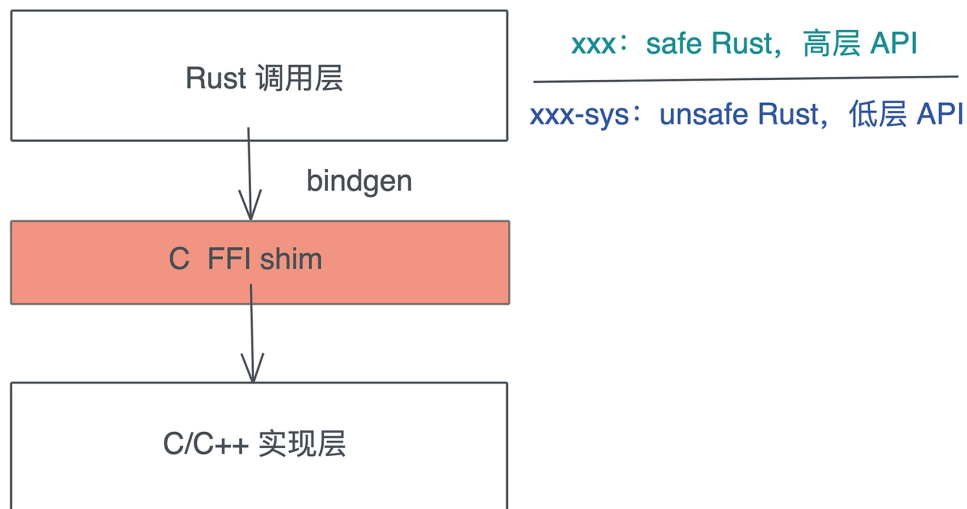
Rust 调用 C 的库

首先看 Rust 和 C/C++ 的互操作。一般而言，当看到一个 C/C++ 库，我们想在 Rust 中使用它的时候，可以先撰写一些简单的 shim 代码，把想要暴露出来的接口暴露出来，然后使用 [bindgen](#) 来生成对应的 Rust FFI 代码。

bindgen 会生成低层的 Rust API，Rust 下约定俗成的方式是**将使用 bindgen 的 crate 命名为 xxx-sys**，里面包含因为 FFI 而导致的大量 unsafe 代码。然后，**在这个基础上生成**

xxx crate，用更高层的代码来封装这些低层的代码，为其它 Rust 开发者提供一套感觉更加 Rusty 的代码。

比如，围绕着低层的数据结构和函数，提供 Rust 自己的 struct / enum / trait 接口。




我们以使用 bindgen 来封装用于压缩 / 解压缩的 bz2 为例，看看 Rust 如何调用 C 的库（以下代码请在 OS X/Linux 下测试，使用 Windows 的同学可以参考 [bzip2-sys](#)）。

首先 cargo new bzip-sys --lib 创建一个项目，然后在 Cargo.toml 中添入：

复制代码

```
1 [dependencies]
2 anyhow = "1"
3
4 [build-dependencies]
5 bindgen = "0.59"
```

其中 bindgen 需要在编译期使用，所以我们在根目录下创建一个 [build.rs](#) 使其在编译期运行：


 复制代码

```

1 fn main() {
2     // 告诉 rustc 需要 link bzip2
3     println!("cargo:rustc-link-lib=bz2");
4
5     // 告诉 cargo 当 wrapper.h 变化时重新运行
6     println!("cargo:rerun-if-changed=wrapper.h");
7
8     // 配置 bindgen, 并生成 Bindings 结构
9     let bindings = bindgen::Builder::default()
10         .header("wrapper.h")
11         .parse_callbacks(Box::new(bindgen::CargoCallbacks))
12         .generate()
13         .expect("Unable to generate bindings");
14
15     // 生成 Rust 代码
16     bindings
17         .write_to_file("src/bindings.rs")
18         .expect("Failed to write bindings");
19 }

```

在 build.rs 里, 引入了一个 wrapper.h, 我们在根目录创建它, 并引用 bzlib.h :

 复制代码


```

1 #include <bzlib.h>

```

此时运行 cargo build, 会在 src 目录下生成 src/bindings.rs, 里面大概有两千行代码, 是 bindgen 根据 bzlib.h 中暴露的常量定义、数据结构和函数等生成的 Rust 代码。感兴趣的话, 你可以看看。

有了生成好的代码, 我们在 src/lib.rs 中引用它:

 复制代码

```

1 // 生成的 bindings 代码根据 C/C++ 代码生成, 里面有一些不符合 Rust 约定, 我们不让编译器报
2 #![allow(non_upper_case_globals)]
3 #![allow(non_camel_case_types)]
4 #![allow(non_snake_case)]
5 #![allow(deref_nullptr)]
6
7 use anyhow::{anyhow, Result};
8 use std::mem;
9
10

```

```

11 mod bindings;
12
    pub use bindings::*;

```

接下来就可以撰写两个高阶的接口 `compress / decompress`，正常情况下应该创建另一个 `crate` 来撰写这样的接口，之前讲这是 Rust 处理 FFI 的惯例，有助于把高阶接口和低阶接口分离。在这里，我们就直接写在 `src/lib.rs` 中：

[复制代码](#)

```

1 // 高层的 API，处理压缩，一般应该出现在另一个 crate
2 pub fn compress(input: &[u8]) -> Result<Vec<u8>> {
3     let output = vec![0u8; input.len()];
4     unsafe {
5         let mut stream: bz_stream = mem::zeroed();
6         let result = BZ2_bzCompressInit(&mut stream as *mut _, 1, 0, 0);
7         if result != BZ_OK as _ {
8             return Err( anyhow!("Failed to initialize") );
9         }
10
11         // 传入 input / output 进行压缩
12         stream.next_in = input.as_ptr() as *mut _;
13         stream.avail_in = input.len() as _;
14         stream.next_out = output.as_ptr() as *mut _;
15         stream.avail_out = output.len() as _;
16         let result = BZ2_bzCompress(&mut stream as *mut _, BZ_FINISH as _);
17         if result != BZ_STREAM_END as _ {
18             return Err( anyhow!("Failed to compress") );
19         }
20
21         // 结束压缩
22         let result = BZ2_bzCompressEnd(&mut stream as *mut _);
23         if result != BZ_OK as _ {
24             return Err( anyhow!("Failed to end compression") );
25         }
26     }
27
28     Ok(output)
29 }
30
31 // 高层的 API，处理解压缩，一般应该出现在另一个 crate
32 pub fn decompress(input: &[u8]) -> Result<Vec<u8>> {
33     let output = vec![0u8; input.len()];
34     unsafe {
35         let mut stream: bz_stream = mem::zeroed();
36         let result = BZ2_bzDecompressInit(&mut stream as *mut _, 0, 0);
37         if result != BZ_OK as _ {
38             return Err( anyhow!("Failed to initialize") );
39         }

```

```
40     // 传入 input / output 进行压缩
41     stream.next_in = input.as_ptr() as *mut _;
42     stream.avail_in = input.len() as _;
43     stream.next_out = output.as_ptr() as *mut _;
44     stream.avail_out = output.len() as _;
45     let result = BZ2_bzDecompress(&mut stream as *mut _);
46     if result != BZ_STREAM_END as _ {
47         return Err(anyhow!("Failed to compress"));
48     }
49
50     // 结束解压缩
51     let result = BZ2_bzDecompressEnd(&mut stream as *mut _);
52     if result != BZ_OK as _ {
53         return Err(anyhow!("Failed to end compression"));
54     }
55 }
56
57 Ok(output)
58 }
59
```

最后，不要忘记了我们的好习惯，写个测试确保工作正常：

[复制代码](#)

```
1  #[cfg(test)]
2  mod tests {
3      use super::*;
4
5      #[test]
6      fn compression_decompression_should_work() {
7          let input = include_str!("bindings.rs").as_bytes();
8          let compressed = compress(input).unwrap();
9          let decompressed = decompress(&compressed).unwrap();
10
11          assert_eq!(input, &decompressed);
12      }
13  }
```

运行 `cargo test`，测试能够正常通过。你可以看到，生成的 [bindings.rs](#) 里也有不少测试，`cargo test` 总共执行了 16 个测试。

怎么样，我们总共写了大概 100 行代码，就用 Rust 集成了 bz2 这个 C 库。是不是非常方便？如果你曾经处理过其他语言类似的 C 绑定，对比之下，就会发现用 Rust 做 FFI 开发真是太方便，太贴心了。

如果你觉得这个例子过于简单，不够过瘾，可以看看 Rust [@RocksDB](#) 的实现，它非常适合你进一步了解复杂的、需要额外集成 C 源码的库如何集成到 Rust 中。

处理 FFI 的注意事项

bindgen 这样的工具，帮我们干了很多脏活累活，虽然大部分时候我们不太需要关心生成的 FFI 代码，但在使用它们构建更高层的 API 时，还是要注意三个关键问题。

如何处理数据结构的差异？

比如 C string 是 NULL 结尾，而 Rust String 是完全不同的结构。我们要清楚数据结构在内存中组织的差异，才能妥善地处理它们。Rust 提供了 [@std::ffi](#) 来处理这样的问题，比如 [@CStr](#) 和 [@CString](#) 来处理字符串。

谁来释放内存？

没有特殊的情况，谁分配的内存，谁要负责释放。Rust 的内存分配器和其它语言的可能不一样，所以，Rust 分配的内存存在 C 的上下文中释放，可能会导致未定义的行为。

如何进行错误处理？

在上面的代码里我们也看到了，C 通过返回的 error code 来报告执行过程中的错误，我们使用了 [@anyhow!](#) 宏来随手生成了错误，这是不好的示例。在正式的代码中，应该使用 thiserror 或者类似的机制来定义所有 error code 对应的错误情况，然后相应地生成错误。

Rust 调用其它语言

目前说了半天，都是在说 Rust 如何调用 C/C++。那么，Rust，调用其他语言呢？

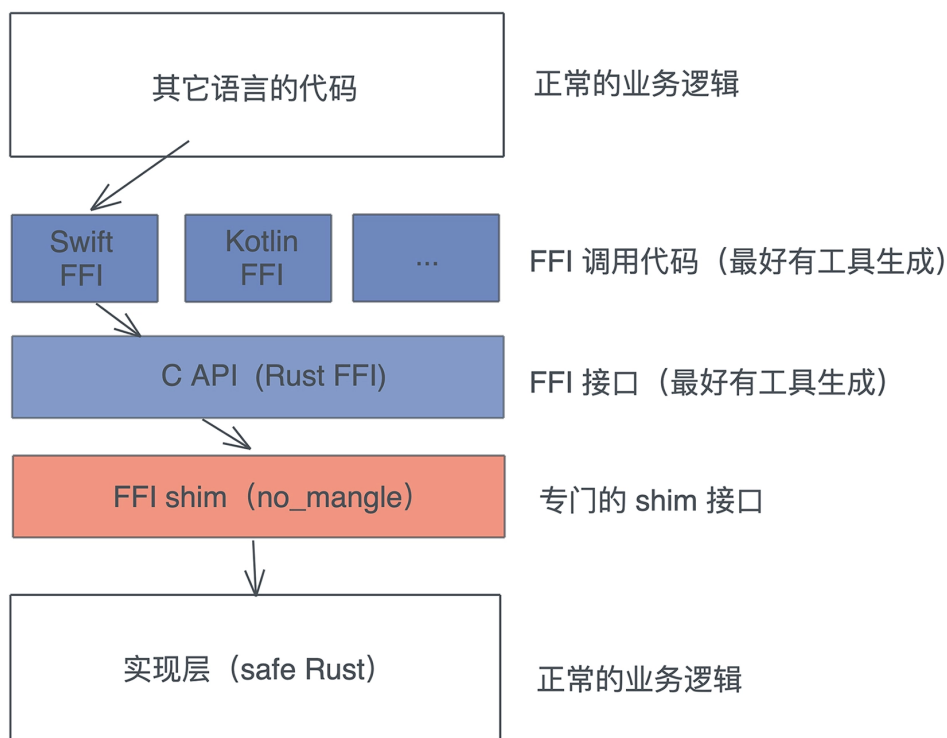
前面也提到，因为 C ABI 深入人心，两门语言之间的接口往往采用 C ABI。从这个角度说，如果我们需要 Rust 调用 Golang 的代码（先不管这合不合理），那么，**首先把 Golang 的代码使用 cgo 编译成兼容 C 的库；然后，Rust 就可以像调用 C/C++ 那样，使用 bindgen 来生成对应的 API 了。**

至于 Rust 调用其它语言，也是类似，只不过像 JavaScript / Python 这样的，与其把它们的代码想办法编译成 C 库，不如把他们的解释器编译成 C 库或者 WASM，然后在 Rust 里调用其解释器使用相关的代码，来的方便和痛快。毕竟，JavaScript / Python 是脚本语言。

把 Rust 代码编译成 C 库

讲完了 Rust 如何使用其它语言，我们再来看看如何把 Rust 代码编译成符合 C ABI 的库，这样其它语言就可以像使用 C 那样使用 Rust 了。

这里的处理逻辑和上面的 Rust 调用 C 是类似的，只不过角色对调了一下：



要把 Rust 代码和数据结构提供给 C 使用，我们首先要构造相应的 Rust shim 层，把原有的、正常的 Rust 实现封装一下，便于 C 调用。

Rust shim 主要做四件事情：

提供 Rust 方法、trait 方法等公开接口的独立函数。注意 C 是不支持泛型的，所以对于泛型函数，需要提供具体的用于某个类型的 shim 函数。

所有要暴露给 C 的独立函数，都要声明成 `#[no_mangle]`，不做函数名称的改写。

如果不用 `#[no_mangle]`，Rust 编译器会为函数生成很复杂的名字，我们很难在 C 中得到正确的改写后的名字。同时，这些函数的接口要使用 C 兼容的数据结构。


数据结构需要处理成和 C 兼容的结构。

如果是你自己定义的结构体，需要使用 `#[repr(C)]`，对于要暴露给 C 的函数，不能使用 `String` / `Vec` / `Result` 这些 C 无法正确操作的数据结构。

要使用 `catch_unwind` 把所有可能产生 `panic!` 的代码包裹起来。

切记，其它语言调用 Rust 时，遇到 Rust 的 `panic!()`，会导致未定义的行为，所以在 FFI 的边界处，要 `catch_unwind`，阻止 Rust 栈回溯跑出 Rust 的世界。

来看个例子：

 复制代码

```
1 // 使用 no_mangle 禁止函数名改编，这样其它语言可以通过 C ABI 调用这个函数
2 #[no_mangle]
3 pub extern "C" fn hello_world() -> *const c_char {
4     // C String 以 "\\0" 结尾，你可以把 "\\0" 去掉看看会发生什么
5     "hello world!\\0".as_ptr() as *const c_char
6 }
```

这段代码使用了 `#[no_mangle]`，在传回去字符串时使用 `"\0"` 结尾的字符串。由于这个字符串在 `RODATA` 段，是 `'static'` 的生命周期，所以将其转换成裸指针返回，没有问题。如果要把这段代码编译为一个可用的 C 库，在 `Cargo.toml` 中，`crate` 类型要设置为 `crate-type = ["cdylib"]`。

刚才那个例子太简单，我们再来看一个进阶的例子。在这个例子里，C 语言那端会传过来一个字符串指针，`format!()` 一下后，返回一个字符串指针：

 复制代码

```
1 #[no_mangle]
2 pub extern "C" fn hello_bad(name: *const c_char) -> *const c_char {
```

```
3     let s = unsafe { CStr::from_ptr(name).to_str().unwrap() };
4
5     format!("hello {}!\0", s).as_ptr() as *const c_char
6 }
```

你能发现这段代码的问题么？它犯了初学者几乎会犯的所有问题。

首先，传入的 `name` 会不会是一个 `NULL` 指针？是不是一个合法的地址？虽然是否是合法的地址我们无法检测，但起码我们可以检测 `NULL`。

其次，`unwrap()` 会造成 `panic!()`，如果把 `CStr` 转换成 `&str` 时出现错误，这个 `panic!()` 就会造成未定义的行为。我们可以做 `catch_unwind()`，但更好的方式是进行错误处理。

最后，`format!("hello {}!\0", s)` 生成了一个字符串结构，`as_ptr()` 取到它堆上的起始位置，我们也保证了堆上的内存以 `NULL` 结尾，看上去没有问题。然而，**在这个函数结束执行时，由于字符串 `s` 退出作用域，所以它的堆内存会被连带 `drop` 掉**。因此，这个函数返回的是一个悬空的指针，在 C 那侧调用时就会崩溃。

所以，正确的写法应该是：

[复制代码](#)

```
1  #[no_mangle]
2  pub extern "C" fn hello(name: *const c_char) -> *const c_char {
3      if name.is_null() {
4          return ptr::null();
5      }
6
7      if let Ok(s) = unsafe { CStr::from_ptr(name).to_str() } {
8          let result = format!("hello {}!", s);
9          // 可以使用 unwrap, 因为 result 不包含 \0
10         let s = CString::new(result).unwrap();
11
12         s.into_raw()
13         // 相当于:
14         // let p = s.as_ptr();
15         // std::mem::forget(s);
16         // p
17     } else {
18         ptr::null()
19     }
20 }
```

在这段代码里，我们检查了 NULL 指针，进行了错误处理，还用 `into_raw()` 来让 Rust 侧放弃对内存的所有权。

注意前面的三个关键问题说过，谁分配的内存，谁来释放，所以，我们还需要提供另一个函数，供 C 语言侧使用，来释放 Rust 分配的字符串：

[复制代码](#)

```
1  #[no_mangle]
2  pub extern "C" fn free_str(s: *mut c_char) {
3      if !s.is_null() {
4          unsafe { CString::from_raw(s) };
5      }
6  }
```

C 代码必须要调用这个接口安全释放 Rust 创建的 CString。如果不调用，会有内存泄漏；如果使用 C 自己的 free()，会导致未定义的错误。

有人可能会好奇，CString::from_raw(s) 只是从裸指针中恢复出 CString，也没有释放啊？

你要习惯这样的“释放内存”的写法，因为它实际上借助了 Rust 的所有权规则：当所有者离开作用域时，拥有的内存会被释放。**这里我们创建一个有所有权的对象，就是为了函数结束时的自动释放。**如果你看标准库或第三方库，经常有类似的“释放内存”的代码。

上面的 hello 代码，其实还不够安全。因为虽然看上去没有使用任何会导致直接或者间接 panic! 的代码，但难保代码复杂后，隐式地调用了 panic!()。比如，如果以后我们新加一些逻辑，使用了 `copy_from_slice()`，这个函数内部会调用 panic!()，就会导致问题。所以，最好的方法是把主要的逻辑封装在 catch_unwind 里：

[复制代码](#)

```
1  #[no_mangle]
2  pub extern "C" fn hello(name: *const c_char) -> *const c_char {
3      if name.is_null() {
4          return ptr::null();
5      }
6
7      let result = catch_unwind(|| {
8          if let Ok(s) = unsafe { CString::from_ptr(name).to_str() } {
9              let result = format!("hello {}!", s);
10             // 可以使用 unwrap, 因为 result 不包含 \\0

```

```
11         let s = CString::new(result).unwrap();
12
13         s.into_raw()
14     } else {
15         ptr::null()
16     }
17 });
18
19 match result {
20     Ok(s) => s,
21     Err(_) => ptr::null(),
22 }
23 }
```

这几段代码你可以多多体会，完整例子放在 [playground](#)。

写好 Rust shim 代码后，接下来就是生成 C 的 FFI 接口了。一般来说，这个环节可以用工具来自动生成。我们可以使用 [cbindgen](#)。如果使用 cbindgen，上述的代码会生成类似这样的 bindings.h：

```
1 #include <stdarg>
2 #include <stdint>
3 #include <stdlib>
4 #include <ostream>
5 #include <new>
6
7 extern "C" {
8
9     const char *hello_world();
10
11     const char *hello_bad(const char *name);
12
13     const char *hello(const char *name);
14
15     void free_str(char *s);
16
17 } // extern "C"
```

[复制代码](#)

有了编译好的库代码以及头文件后，在其他语言中，就可以用该语言的工具进一步生成那门语言的 FFI 绑定，然后正常使用。

和其它语言的互操作

好，搞明白 Rust 代码如何编译成 C 库供 C/C++ 和其它语言使用，我们再看看具体语言有没有额外的工具更方便地和 Rust 互操作。

对于 Python 和 Node.js，我们之前已经见到了 [PyO3](#) 和 [Neon](#) 这两个库，用起来都非常简单直观，下一讲会再深入使用一下。

对于 Erlang/Elixir，可以使用非常不错的 [rustler](#)。如果你对此感兴趣，可以看这个 [repo](#) 中的演示文稿和例子。下面是一个把 Rust 代码安全地给 Erlang/Elixir 使用的简单例子：

[复制代码](#)

```
1 #[rustler::nif]
2 fn add(a: i64, b: i64) -> i64 {
3     a + b
4 }
5
6 rustler::init!("Elixir.Math", [add]);
```

对于 C++，虽然 cbindgen 就足够，但社区里还有 [cxx](#)，它可以帮助我们很方便地对 Rust 和 C++ 进行互操作。

如果你要做 Kotlin / Swift 开发，可以尝试一下 mozilla 用在生产环境下的 [uniffi](#)。使用 uniffi，你需要定义一个 UDL，这样 uniffi-bindgen 会帮你生成各种语言的 FFI 代码。

具体怎么用可以看这门课的 [GitHub repo](#) 下这一讲的 ffi-math crate 的完整代码。这里就讲一下重点，我写了个简单的 [uniffi 接口](#) (math.udl)：

[复制代码](#)

```
1 namespace math {
2     u32 add(u32 a, u32 b);
3     string hello([ByRef]string name);
4 };
```

并提供了 Rust 实现：

```
1 uniffi_macros::include_scaffolding!("math");
2
3 pub fn add(a: u32, b: u32) -> u32 {
4     a + b
5 }
6
7 pub fn hello(name: &str) -> String {
8     format!("hello {}!", name)
9 }
```

[复制代码](#)

之后就可以用：

```
1 uniffi-bindgen generate src/math.udl --language swift
2 uniffi-bindgen generate src/math.udl --language kotlin
```

[复制代码](#)

生成对应的 Swift 和 Kotlin 代码。

我们看生成的 hello() 函数的代码。比如 Kotlin 代码：

```
1 fun hello(name: String): String {
2     val _retval =
3         rustCall() { status ->
4             _UniffiLib.INSTANCE.math_6c3d_hello(name.lower(), status)
5         }
6     return String.lift(_retval)
7 }
```

[复制代码](#)

再比如 Swift 代码：

```
1 public func hello(name: String) -> String {
2     let _retval = try!
3
4     rustCall {
5         math_6c3d_hello(name.lower(), $0)
6     }
7     return try! String.lift(_retval)
8 }
```

[复制代码](#)

你也许注意到了这个 `RustCall`，它是用来调用 Rust FFI 代码的，看源码：

[复制代码](#)

```

1 private func rustCall<T>(_ callback: (UnsafeMutablePointer<RustCallStatus>) ->
2     try makeRustCall(callback, errorHandler: {
3         $0.deallocate()
4         return UniffiInternalError.unexpectedRustCallError
5     })
6 }
7
8 private func makeRustCall<T>(_ callback: (UnsafeMutablePointer<RustCallStatus>
9     var callStatus = RustCallStatus()
10    let returnedVal = callback(&callStatus)
11    switch callStatus.code {
12    case CALL_SUCCESS:
13        return returnedVal
14
15    case CALL_ERROR:
16        throw try errorHandler(callStatus.errorBuf)
17
18    case CALL_PANIC:
19        // When the rust code sees a panic, it tries to construct a RustBuffer
20        // with the message. But if that code panics, then it just sends back
21        // an empty buffer.
22        if callStatus.errorBuf.len > 0 {
23            throw UniffiInternalError.rustPanic(try String.lift(callStatus.err
24        } else {
25            callStatus.errorBuf.deallocate()
26            throw UniffiInternalError.rustPanic("Rust panic")
27        }
28
29    default:
30        throw UniffiInternalError.unexpectedRustCallStatusCode
31    }
32 }
```

你可以看到，它还考虑了如果 Rust 代码 panic! 后的处理。那么 Rust 申请的内存会被 Rust 释放么？

会的。hello() 里的 `String.lift()` 就在做这个事情，我们看生成的代码：

[复制代码](#)

```

1 extension String: ViaFfi {
2     fileprivate typealias FfiType = RustBuffer
```

```
3
4     fileprivate static func lift(_ v: FfiType) throws -> Self {
5         defer {
6             v.deallocate()
7         }
8         if v.data == nil {
9             return String()
10        }
11        let bytes = UnsafeBufferPointer<UInt8>(start: v.data!, count: Int(v.le
12        return String(bytes: bytes, encoding: String.Encoding.utf8)!
13    }
14    ...
15 }
16
17 private extension RustBuffer {
18     ...
19     // Frees the buffer in place.
20     // The buffer must not be used after this is called.
21     func deallocate() {
22         try! rustCall { ffi_math_6c3d_rustbuffer_free(self, $0) }
23     }
24 }
```

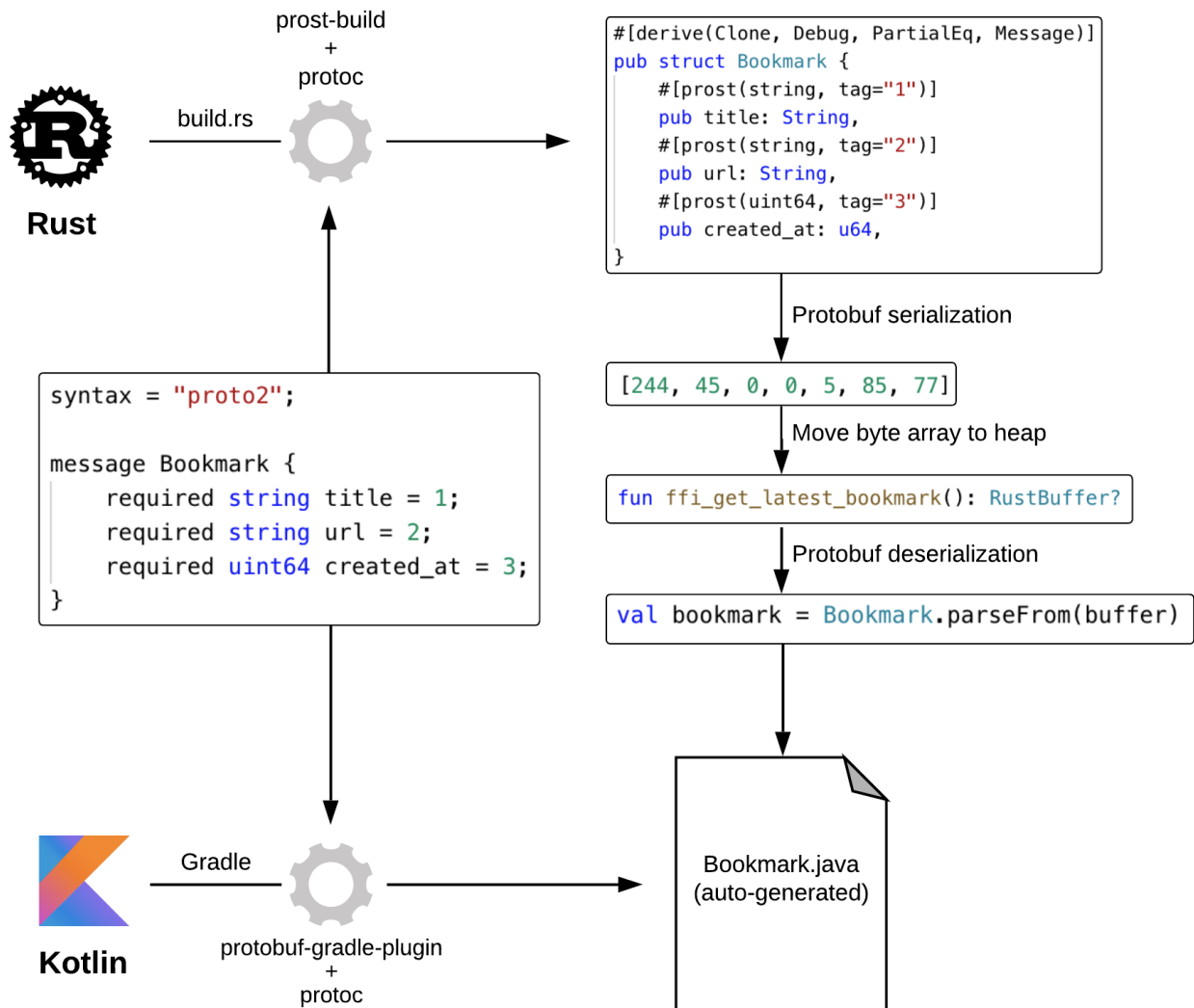
在 lift 时，它会分配一个 swift String，然后在函数退出时调用 deallocate()，此时会发送一个 rustCall 给 ffi_math_rustbuffer_free()。

你看，uniffi 把前面说的处理 FFI 的三个关键问题：**处理数据结构的差异、释放内存、错误处理**，都妥善地解决了。所以，如果你要在 Swift / Kotlin 代码中使用 Rust，非常建议你使用 uniffi。此外，uniffi 还支持 Python 和 Ruby。

FFI 的其它方式

最后，我们来简单聊一聊处理 FFI 的其它方式。其实代码的跨语言共享并非只有 FFI 一条路子。你也可以使用 REST API、gRPC 来达到代码跨语言使用的目的。不过，这样要额外走一圈网络，即便是本地网络，也效率太低，且不够安全。有没有更高效一些的方法？

有！我们可以在两个语言中使用 protobuf 来序列化 / 反序列化要传递的数据。在 Mozilla 的一篇博文 [Crossing the Rust FFI frontier with Protocol Buffers](#)，提到了这种方法：



感兴趣的同学，可以读读这篇文章。也可以看看我之前写的文章 [深度探索：前端中的后端](#)，详细探讨了把 Rust 用在客户端项目中的可能性以及如何做 Rust bridge。

小结

FFI 是 Rust 又一个处于领先地位的领域。

从这一讲的示例中我们可以看到，在支持很方便地使用 C/C++ 社区里的成果外，Rust 也可以非常方便地在很多地方取代 C/C++，成为其它语言使用底层库的首选。**除了方便的 FFI 接口和工具链，使用 Rust 为其它语言提供底层支持，其实还有安全性这个杀手锏。**

比如在 Erlang/Elixir 社区，高性能的底层 NIF 代码，如果用 C/C++ 撰写的话，一个不小心就可能导致整个 VM 的崩溃；但是用 Rust 撰写，因为其严格的内存安全保证（只要保证 unsafe 代码的正确性），NIF 不会导致 VM 的崩溃。

所以，现在 Rust 越来越受到各个高级语言的青睐，用来开发高性能的底层库。

与此同时，当需要开发跨越多个端的公共库时，使用 Rust 也会是一个很好的选择，我们在前面的内容中也看到了用 uniffi 为 Android 和 iOS 构建公共代码是多么简单的一件事。

思考题

1. 阅读 [std::ffi](#) 的文档，想想 `Vec<T>` 如何传递给 C？再想想 `HashMap<K,V>` 该如何传递？有必要传递一个 `HashMap` 到 C 那一侧么？
2. 阅读 [rocksdb](#) 的代码，看看 Rust 如何提供 rocksDB 的绑定。
3. 如果你是个 iOS/Android 开发者，尝试使用 Rust 的 `reqwest` 构建 REST API 客户端，然后把得到的数据通过 FFI 传递给 Swift/Kotlin 侧。

感谢你的收听，今天完成了第 31 次 Rust 学习打卡啦。如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见～

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 7  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | Unsafe Rust：如何用 C++ 的方式打开 Rust？

下一篇 32 | 实操项目：使用 PyO3 开发 Python3 模块

精选留言 (4)

 写留言



Marvichov 

2021-11-16

> The ABI for C is platform-specific (OS, processor) - it covers issues such as register allocation and calling conventions, which are obviously specific to a particular processor.

<https://stackoverflow.com/questions/4489012/does-c-have-a-standard-abi>

...

展开 ∨



Marvichov

2021-11-14

私以为对FFI的理解, 重点还是对ABI的理解;

C-ABI就像英语一样...不同母语的人可以通过英语交流...数据转换就相当于翻译的过程...中文 → 英文 → 法文; 目前很多机器翻译AI也是把target lang翻译成英语...英语有点像一个MIR了...

展开 ∨

共 1 条评论 >



Marvichov

2021-11-14

Q: 有个小问题, 为啥bindings.h不需要以下这些header, 咋一build就自动添加这些header呢? 难道是ffi的scaffolding的代码需要?

```
#include <csdarg>
#include <csdint>...
```

展开 ∨



罗杰

2021-11-08

如何在 build.rs 断点调试呢?

展开 ∨

作者回复: 没有试过, 需要找到 build binary 对应的 process, 然后想办法 gdb attach 进去。这里有个讨论: https://www.reddit.com/r/rust/comments/72ip1h/attach_gdb_to_buildrs/

