# Racket 调用 Rust 的若干想法

荀涧林

二〇二三年七月十五日

## 1 背景

在准备好一门自用的方言（azelf）后，准备写点实际应用，第一个目标就是写爬虫。racket 本身有一个 sxml 库，但文档语焉不详、接口不明，远不如市面上正常的解析库方便，同时也不支持 css selector，写爬虫造成诸多不便。

Rust 作为一门新生底层语言，与 C++、C 比较，较少（完全没有）历史包袱，完善的文档及强大的包管理，同时具备完整的 FFI 能力，很自然它便成了目标对象。

## 2 基本环境

不必担心太过复杂的项目配置，racket 和 rust 都自己的包（项目）管理器，而且我们都不用任何外部依赖，仅靠标准库就能实现交互功能。

我们用命令依次建立两个项目，不妨都叫作 ffi-sample，项目地址可以任意，不要忘记就行：

```
# 在任意地方建立rust项目
> cargo new --lib ffi-sample
# 在任意地方建立racket项目
> raco pkg new ffi-sample
```

这只完成了项目初始，我们需要稍作修改，进入 rust 项目，在 cargo.toml 添加：

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

指定项目要生成 C 动态链接库。待动态库生成后，racket 那边就可以作链接了，我们可以先将 racket 项目修改一下：

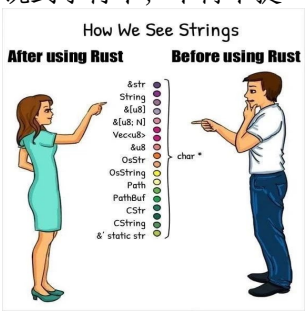
```
(require ffi/unsafe
         ffi/unsafe/define)

(define-ffi-definer
  define-rust
  (ffi-lib (expand-user-path "<the path of rust ffi-sample>/target/debug/libffi_sample.so")))

(define-rust say-hello
```

```
  (_fun -> _void)
  #:c-id say_hello)
```

`define-ffi-definer`是动态库的绑定,绑定后会生成一个宏,在这里例子里,我们将这个宏命名为 define-rust。利用该宏，可以将 racket 函数与 rust 导出的函数绑定，例如上面的例子里，我们将 rust 的`say_hello`绑定到 say-hello 上。

# 3 传递字符串

说到字符串，不得不提一下这张图：




在 Rust 里，常用的是 str、String，为了将它们传给 racket，我们可以写个测试代码：

```rust
#[no_mangle]
pub extern "C" fn tiny_str() -> *const c_char {
  let s: &str = "hello string";
  s.as_ptr().cast()
}
```

Rust 代码

```racket
(define-rust tiny-str
(_fun -> _string)
#:c-id tiny_str)

(displayln (tiny-str))
```

Racket 代码

正常显示，看来还是能用的。

## 3.1 转移所有权

上面代码运行良好，我们继续改造一下 rust 代码，将 str 换成 String 会怎样。

```rust
#[no_mangle]
pub extern "C" fn tiny_str() -> *const c_char {
    let s = String::from("hello string");
    s.as_ptr().cast()
}
```

看起来跟之前没多大不同，我们再运行一下 racket 代码，结果出人意料，崩溃了。

```
bytes->string/utf-8: byte string is not a well-formed UTF-8 encoding
  byte string: #"\361\336\212U\5"
```

出错信息

这是因为 &str 是全局的，离开 tiny_str 后，并不会回收。String 是分配到堆上，由于 Rust 的生命周期的管理，离开 tiny_str，触发析构回收，导致最后得到一个已经释放过资源的悬挂指针，也就导致了 racket 无法正确解析。

为了解决这个问题，rust 标准库里提供了`std::mem::forget`，不会强制执行析构函数，我们也就能够把资源完整地转移出去了。

```rust
#[no_mangle]
pub extern "C" fn tiny_str() -> *const c_char {
    let s = String::from("hello string");
    let ptr = s.as_ptr();
    std::mem::forget(s);
    ptr.cast()
}
```

## 3.2 C 风格字符串与内存泄漏

我们把 String 转移了出去，但我们再也没有能力简易地收回来，安全的 rust 没有提供 *const c_char 这样的方法，当然，unsafe 里也没有。

rust 和 racket 都不是 C 风格的'\0' 中断字符串，我们以 String 为例，当 rust 把整个 String 指针传出时，racket 会直接将内存深拷贝一价，最终生成 racket 本身的 string，不受制于中断符。稍微细心一点就能发现，racket 是深拷贝生成 string，那么 rust 转移出来的 String 的所有权到哪里去了呢？答案很不幸，此时的 String 已经变成野指针，它永远也不会被释放，所以只管生产，不管回收，最终的结局就是内存泄漏！

我们讲 FFI，一般都是讲 C 的 ABI 规范与兼容性，虽然 racket 和 rust 的 string 能相通，从规范来讲，我们最好能提供一份 C 兼容的字符串，用于两者交互。rust 标准库里就提供了 CStr 和 CString，相对于 str 和 String。我们讲到了转移字符器所有权会有内存泄漏的风险，所以我们除了要写一个生成字符串的函数，还要有一个相配套的释放函数。

```rust
#[no_mangle]
pub extern "C" fn get_cstring() -> *const c_char {
  CString::new("hello world").unwrap().into_raw()
}

#[no_mangle]
pub extern "C" fn free_cstring(ptr: *mut c_char) {
  unsafe { CString::from_raw(ptr) };
}
```

Rust 代码

```racket
(define _cstring (_cpointer 'cstring))

(define-rust get-cstring
  (_fun -> _cstring)
  #:c-id get_cstring)

(define-rust free-cstring
  (_fun _cstring -> _void)
  #:c-id free_cstring)

(define (safe-cstring)
  (define ptr (get-cstring))
  (define s (cast ptr _cstring _string))
  (free-cstring ptr)
  s)
```

Racket 代码

rust 代码很简单，CString::into_raw 能够转移所有权到外部调用者，而且不会在离开作用域后提交释放资源；CString::from_raw 就是用来接受 CString::into_raw 的指针，接管整个生命周期，当它离开作用域时自动析构，所以我们在代码上不需要显示调用 drop。racket 那一侧，就需要手动释放：先得到 CString 指针 ptr，再 cast 成默认的 string（深拷贝了一次），ptr 的内容依然完整，最后调用 free-cstring 释放。

# 4 传递自定义结构体

从一个语言的类型，传递到另一个语言的类型，一般会伴随着深拷贝，如果为了节约更多资源，那么就要减少不必要的拷贝，或许可以让调用者直接操作指针，减少转化的成本。

假如说我们有这样的类型：User，它只包含姓名（String）和年龄（u64），我们需要构造这样的用户，rust 代码如下：

```rust
#[derive(Debug)]
pub struct User {
    pub name: String,
    pub age: u64,
}

#[no_mangle]
pub extern "C" fn create_user(name_ptr: *const c_char, age: u64) -> *mut User {
    let name = unsafe { CStr::from_ptr(name_ptr) };
    let name = name.to_str().unwrap().into();
    Box::into_raw(Box::new(User { name, age }))
}

#[no_mangle]
pub extern "C" fn show_user(ptr: *mut User) {
    let user = unsafe { Box::from_raw(ptr) };
    println!("{:?}", user);
}
```

User 整个定义，除了 age 是 C 兼容，其它（也就一个 name）字段都是 rust 标准类型，无非与外部语言互操。这样的字段一多，每次转化都要经历一次深拷贝，这样的开销可能承受不住，倒不如一了百了，我们直接传出 User 指针，不再直接转化，同时我们提供函数（例如 show_user）用于相关操作。

```racket
(define _user_ptr (_cpointer 'user))

(define-rust create-user
  (_fun _string _uint64 -> _user_ptr)
  #:c-id create_user)

(define-rust show-user
  (_fun _user_ptr -> _void)
  #:c-id show_user)
```

Racket 这边就简单了，直接保存指针就行，需要使用时再传回去。

## 4.1 接受所有权还是引用?

我们在 racket 侧运行一下代码，看是否正常。

```
(define ptr (create-user "name" 10))
(show-user ptr)
```

结果很正常。这两行代码目前为止看起来很正常，但再我们继续 show-user，结果就出人意料了:

```
invalid memory reference
```

提示只有这么几句，但也基本表明了错误原因，原因就在于释放了两次！show-user 这个函数写得有问题，从功能上讲，它仅仅为了打印用户信息，不该自作主张做释放。看代码，很明显出在 Box::from_raw 身上，它跟 CString::from_raw 一样接管了整理生命周期，它在打印之后，离开函数，结束了自己。我们当然可以继续用`std::mem::forget`强制不去释放，但请仔细想想，我们本不该拿到它的所有权，我们需要的仅仅是它的引用，这里用 Box::from_raw 是不合适的。既然是引用，打出 &* 就能拿到。

```
#[no_mangle]
pub extern "C" fn show_user(ptr: *const User) {
    let user = unsafe { &*ptr };
    println!("{:?}", user);
}

#[no_mangle]
pub extern "C" fn free_user(ptr: *mut User) {
    unsafe { Box::from_raw(ptr) };
}
```

show_user 只拿引用，free_user 才需要获取它的所有权。接下去，show_user 不论调用多少次都不会有问题。

## 4.2 RAII

上面两个例子可以看到，每次从 rust 申请一次，都要手动释放一次，写起来十分不方便，对于有 GC 的语言来讲，写起来跟无 GC 一样，十分痛苦。好在 racket 提供了`wrap`。

```
(define-rust free-user
  (_fun _user_ptr -> _void)
  #:c-id free_user
  #:wrap (deallocator))

(define-rust create-user
  (_fun _string _uint64 -> _user_ptr)
  #:c-id create_user
  #:wrap (allocator free-user))

(define ptr (create-user "name" 10))
```

```
(show-user ptr)
```

deallocator 和 allocator 在 ffi/unsafe/alloc 中，总的来说算是一种定式，按这种格式去写就行。show-user 无论调用多少次都不会内存泄漏，racket 已经做好资源管理了，只管放心调用。

# 5 传递数组

依照先前说法，类型间的转化都会伴随着拷贝，那我们依然传出指针，然后再提供 FFI 函数用于互操。

```
#[no_mangle]
pub extern "C" fn create_n_vec(n: u64) -> *const Vec
    <u64> {
    let xs: Vec<_> = (1..=n).collect();
    Box::into_raw(Box::new(xs))
}

#[no_mangle]
pub extern "C" fn create_cstr_vec(n: u64) -> *const
    Vec<String> {
    let xs: Vec<_> = (1..=n).map(|i| i.to_string()).
        collect();
    Box::into_raw(Box::new(xs))
}

#[no_mangle]
pub extern "C" fn vec_len(ptr: *const Vec<c_void>)
    -> usize {
    let xs = unsafe { &*ptr };
    xs.len()
}
```

```
(define _vec_ptr (_cpointer 'vec))
(define-rust free-cstr-vec
  (_fun _vec_ptr -> _void)
  #:c-id free_cstr_vec
  #:wrap (deallocator))

(define-rust create-n-vec
  (_fun _uint64 -> _vec_ptr)
  #:c-id create_n_vec
  #:wrap (allocator free-n-vec))

(define-rust vec-len
  (_fun _vec_ptr -> _uint64)
  #:c-id vec_len)
```

## 5.1 不算泛型的泛型接口

只要不涉及 Vec 元素本身，我们还是能够写出一些通用的接口，用于查询 Vec 本身的状态，如长度、是否为空。

```
#[no_mangle]
pub extern "C" fn vec_len(ptr: *const Vec<c_void>) -> usize {
    let xs = unsafe { &*ptr };
    xs.len()
}
```

我们用了 c_void 类型，因为 FFI 函数不允许泛型，从 C 角度来讲，void* 就是另一种泛型表示。我们把上述两种不同的数组传进去，发现是能正常运行的。

```
(vec-len (create-n-vec 10))
(vec-len (create-cstr-vec 20))
```

## 5.2 迭代数组之回调

除了普通值，我们还可以传回调函数进去，能传函数，在某些方面能简化的开发，我们依然以数组为例。

```rust
#[no_mangle]
pub extern "C" fn iter_vec(ptr: *const Vec<u64>, f:
    extern "C" fn(u64)) {
    let xs = unsafe { &*ptr };
    xs.iter().for_each(|x| f(*x));
}
```

```racket
(define-rust iter-vec
  (_fun _vec_ptr (_fun _uint -> _void) -> _void)
  #:c-id iter_vec)
(define n-ptr (create-n-vec 10))
(iter-vec n-ptr displayln)
```

可以看到结果正常输出。虽然能调用回调，但在此处的回调函数没什么大用，在纯函数眼里，它仅仅处理了列表中的元素，无法将它们组合起来，作为新数组传出。

## 5.3 迭代数组之按位取值

我们不再往纯函数方面考虑，我们以传统的 C 风格遍历。

```rust
#[no_mangle]
pub unsafe extern "C" fn iter_index(ptr: *const Vec<
    u64>, index: usize) -> u64 {
    let xs = &*ptr;
    *xs.get_unchecked(index)
}
```

```racket
(define-rust iter-index
  (_fun _vec_ptr _uint -> _uint)
  #:c-id iter_index)
```

我们仅添加 iter-index 函数，用于取出某位元素，然后就可以利用 for 循环，遍历整个元素：

```racket
(define n-ptr (create-n-vec 10))
(define len (vec-len n-ptr))
(->> (for/list ([n (in-range len)])
       (iter-index n-ptr n))
     displayln)
```

最后生成了 racket 的 list。

# 6 总结

上面总结了 racket 调用 rust 若干方法及相关想法，一方面要保证资源的安全性，另一方面也要保证一定的性能，不能到处拷贝。

racket 作为老牌语言，设计上考虑了很多方面，作为一门教学语言，却拥有了大部分工业需要的功能；rust 作为新兴语言，它肯定是合格的现代语言，不管是现代的类型系统，还是人性化的包管理，甚至还配置了文档功能，这在 cabal 也是少见的，至少我在用 stack 时才能看到。

不管如何，这两个很有意思的语言，如今能将它们打通，racket 加上 rust 的生态，感觉也能做出更多有意思的事。