

## • 前言

"FFI"是" [Foreign Function Interface](#)"的缩写，大意为不同编程语言所写程序间的相互调用。鉴于C语言事实上是编程语言界的万国通，世界通用语，所以本文主要围绕着C和Rust之间的互通来学习。

单刀直入，话不啰嗦，好比学外语，先要从认字开始，对于编程语言来说就是各种“基础类型”，因为类型代表了：可操作集和布局，有人会疑问“类型布局”是个什么东西？！好吧，换个词“房屋布局”，这词的意思，您好理解吧！对！代表了：位置、大小、方向、排列、顺序等信息！在类型的基础上，进一步考虑：类型的组合(打包封装对齐)方式，这也好理解吧！毕竟人与人沟通光蹦字还不行，字组合起来才有意思呀！再下一步就是“函数声明”，代表着想去干一件事情！首先要描述清楚：比如：函数类型、输入参数、结果类型、函数调用规约（如：\_\_stdcall、\_cdecl等等好多，代表函数栈的使用和清理、函数调用参数的入栈顺序、排列方式，函数返回指针等），下一步内存管理问题，彼此互传的数据对象该如何划分和管理其生命周期！避免出现“悬指针和泄露”，最后还要有回调函数或闭包之类用于状态通知！好比好朋友，别什么事都等我自己主动去问，你最好时不时主动告诉我呀！

当然将上面两种编程语言编写的代码统一管理起来，还需要相应机制，解决诸如：编译构建，库生成和引用等问题。

此篇Rust FFI文章比较全面：[`https://doc.rust-lang.org/nomicon/ffi.html`](https://doc.rust-lang.org/nomicon/ffi.html)

## • 基础类型

每一种编程语言都可能定义许多“基础类型”，两种编程语言的基础类型之间最好有一个交集，这样才能传递数据，所以：Rust `std::ffi` 和 The libc crate就是非常重要的C and Rust的基础类型交集，

它俩是语言互通的一个前提基础，Rust `std::ffi`模块提供了诸如：[c\\_void](#)、

[CString](#)、[CStr](#)、[OsString](#)、[OsStr](#)等和Rust自己的字符串类型：String 和str 之间的转化类型。详情请参看：[`https://doc.rust-lang.org/std/ffi/`](https://doc.rust-lang.org/std/ffi/)。而the libc crate 则

封装了更加丰富的C数据类型和API，诸如：[c\\_void](#)、`c_char`、`c_float`、`c_double`、

`c_int`、`c_long`、`c_schar`、`c_uint`等，详情请参看：[`https://docs.rs/libc/0.2.70/libc/`](https://docs.rs/libc/0.2.70/libc/)

。``std::os::raw`也同样定义了一些C基础类型`，这3者存在重叠交集，一般使用前两个就足矣。

## • C and Rust 混合工程管理

Rust 不支持源代码级别直接与其他语言的交互， 也就是说不支持直接在 Rust code 中直接 embed 内嵌其他编程语言代码！只支持以二进制库的方式来互相调用，所以语言界限清晰明确，

避免诸多问题。当然有第三方 crate 以宏的方式间接支持了在 Rust code 中内嵌其他语言代码， 详情请参看：<https://github.com/mystor/rust-cpp> 。

### (1) Rust 支持的库类型：

- `lib` — Generates a library kind preferred by the compiler, currently defaults to `rlib`.
- `rlib` — A Rust static library.
- `staticlib` — A native static library.
- `dllib` — A Rust dynamic library.
- `cdllib` — A native dynamic library.
- `bin` — A runnable executable program.
- `proc-macro` — Generates a format suitable for a procedural macro library that may be loaded by the compiler.

注意： 其中 `cdllib` 和 `staticlib` 就是与 C ABI 兼容的。

### (2) Build C code to a static or dynamic library

```
1 #全手动构建C动态库on Linux with gcc
2 #gcc -Wall -fPIC -c a.c
3 #gcc -shared -o libtest.so a.o
4 #-----
5 #全手动构建C静态库on Linux with gcc
6 #gcc -c b.c
7 #gcc -c a.c
8 #ar rcs libtest.a a.o b.o
9 #-----
10 #gcc option: -I 查找头文件位置, -L 查找库位置, -l 小写l指示库名字
```

### (3) Rust call C [rust 工程位置:

`rust_learn/unsafe_collect/ffi/rust2c/manual_build_1]`

```
1 //manual_build_1工程目录树形结构
```

```
2 .
3 |— Cargo.lock
4 |— Cargo.toml
5 |— src
6 |   └─ main.rs
7 └─ test.c
8
9 1 directory, 4 files
```

```
1 //test.c
2 //下面的shell 命令行用于编译构建一个c static
3 // gcc -c -Wall -Werror -fpic test.c //for dynamic library
4 //gcc -c test.c //for static library.
5 //ar rcs libtest.a test.o //for static library
6
7 int add(int a, int b) {
8     return a +b;
9 }
```

```
1 //RUSTFLAGS='-L .' cargo run
2 //-L 用于告诉rustc 库位置。
3
4 use std::os::raw::c_int; //(1) 必须使用rust and c都认识的数据类型。
5
6 //(2) 这个rust属性用于按名指定链接库，默认链接动态库，除非kind设定static指定链接静态库。
7 //相当于 -l 的作用
8 //rustc 发现动态库没找到，它就自动去找静态库， 只要名字相同。
9 #[link(name="test")]
10 //#[link(name = "test", kind = "static")]
11
12 //(3) 申明外部函数遵守C语言函数调用规约。
13 extern "C" {
14     fn add(a: c_int, b: c_int) -> c_int; //此声明函数实际定义在C库中，由上面link属性指定。
15 }
16
17 fn main() {
18     //(4) Rust 规定，只允许在unsafe块中调用FFI extern fn.
19     let r = unsafe{add(2, 3)};
20     println!("{}", r);
21 }
22
```

## (4) 向rustc 传参的几种方法

(a) rustc -L 指定库搜索位置 -l 库名

(b) `RUSTFLAGS='-L my/lib/location'` `cargo build` # or `cargo run`

(c) `rustc-link-search` 相当于`-L` , 具体解释看下面代码例子

```
1 # 编辑Cargo.toml, 指定启用build.rs 用于在开始构建rust code之前首先执行, 构建好各种依赖环境, 如提前构建好C库。
2 [package]
3 name = "link-example"
4 version = "0.1.0"
5 authors = ["An Devloper <an.developer@example.com>"]
6 build = "build.rs" #关键点
```

```
1 //编辑build.rs
2 fn main() {
3     //关键就是这个println!, 将rustc-link-search设定为我们自己实际的C库路径就好。
4     println!(r"cargo:rustc-link-search=库的位置目录");
5 }
```

**【更多方法请您参看: <https://stackoverflow.com/questions/40833078/how-do-i-specify-the-linker-path-in-rust> 和 <https://doc.rust-lang.org/cargo/reference/build-script-examples.html>】**

以及<https://doc.rust-lang.org/cargo/reference/build-scripts.html#outputs-of-the-build-script>和<https://doc.rust-lang.org/cargo/reference/environment-variables.html>

**上面的rust工程例子只是通过手动一个个敲命令来构建的, 十分繁琐, 只适用于展示原理, 实际工程不可取。下面开始研究几个自动完成C and Rust 工程编译构建的例子。**

```
1 #下面是build_c_lib_by_gcc工程目录树形结构, 里面包含了C代码文件。
2 .
3 └─ Cargo.toml
4 └─ build.rs
5   └─ src
6     └─ hello.c
7       └─ main.rs
8
9 1 directory, 4 files
```

```
1 #配置Cargo.toml
2 [package]
3 name = "build_c_lib_by_gcc"
4 version = "0.1.0"
5 authors = ["yujinliang <285779289@qq.com>"]
6 edition = "2018"
7 build="build.rs" #关键点, 启用构建脚本build.rs。
8 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
```

```
9
10 [dependencies]
11
```

```
1 // build.rs
2
3 use std::process::Command;
4 use std::env;
5 use std::path::Path;
6
7 fn main() {
8     let out_dir = env::var("OUT_DIR").unwrap();
9
10    //下面直接调用gcc生成C库，并未考虑跨平台问题，切切！
11    Command::new("gcc").args(&["src/hello.c", "-c", "-fPIC", "-o"])
12        .arg(&format!("{}/hello.o", out_dir))
13        .status().unwrap();
14    Command::new("ar").args(&["crus", "libhello.a", "hello.o"])
15        .current_dir(&Path::new(&out_dir))
16        .status().unwrap();
17    //上面的代码很直观，就是编译C 代码，构建静态库的命令行， 生成的C库存放到"OUT_DIR"环境变量
    指定的目录。
18    //其实您完全可以举一反三， 通过编写build.rs构建脚本，可以调用诸如gcc, ar, make,cmake等
    C/C++构建工具为Rust工程提前生成C库。
19    //我想您能想到， build.rs肯定是在开始构建编译Rust工程之前执行的！用于预处理。
20
21    //下面很关键，配置cargo的官方指定方法之一 ！
22    println!("cargo:rustc-link-search=native={}", out_dir); //配置C库的搜索路径，相当于rustc -L
23    println!("cargo:rustc-link-lib=static=hello"); //配置需要链接的C库名，相当于rustc -l
24    println!("cargo:rerun-if-changed=src/hello.c"); //告诉cargo工具，只有当“src/hello.
    c”这个文件发生变化时，才重新执行build.rs脚本。
25 }
```

```
1 //src/main.rs
2 //注意：此处没有使用#[link]属性指定需要链接的C库， 因为我们在build.rs构建脚本中已经设定好
    了，
3 //rust cargo 知道该去链接那个C库。
4 extern "C" { fn hello(); }
5
6 fn main() {
7     unsafe { hello(); }
8 }
```

```
1 // src/hello.c
2
```

```

3 #include <stdio.h>
4
5 void hello() {
6     printf("Hello, World!\n");
7 }

```

the cc crate可以帮助我们自动处理构建编译C/C++库的繁琐过程，同时自动检测平台和架构，自动选择编译器，构建工具，设定好编译参数，设定好相关cargo 指令和环境变量等，高效简洁，下面我们看看例子。

```

1 #cc_auto_build_c_lib工程目录结构
2 .
3 └─ build.rs
4 └─ Cargo.lock
5 └─ Cargo.toml
6 └─ src
7   └─ hello.c
8   └─ main.rs
9
10 1 directory, 5 files

```

```

1 [package]
2 name = "cc_auto_build_c_lib"
3 version = "0.1.0"
4 authors = ["yujinliang <285779289@qq.com>"]
5 edition = "2018"
6 build="build.rs" #启用build.rs构建脚本。
7 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
8
9 [build-dependencies] #用于配置build.rs用到的各种依赖项。
10 cc = "1.0.53" #自动构建编译C/C++代码。
11
12 [dependencies]

```

```

1 //build.rs
2 fn main() {
3     //the cc crate专门自动构建编译C/C++ code,
4     //如：自动检测：系统平台， 硬件架构， 自动选择相应编译器，设定各种编译参数，
5     //自动设定相关环境变量， 如：cargo相关环境变量， 自动将编译好的C库保存到“OUT_DIR”
6     //所以cc可以自动帮你搞定诸如：交叉编译， 跨平台。
7     //cargo build -vv 可以看到已经自动设定的各种构建参数。
8     //详情请参考：`https://docs.rs/cc/1.0.53/cc/`
9     cc::Build::new()
10     .file("src/hello.c")
11     .compile("hello");
12     println!("cargo:rerun-if-changed=src/hello.c"); //告诉cargo 只有当src/hello.c发生变化时，才重新执行build.rs脚本。

```

```
13 }
```

```
1 //src/main.rs
2 //注意：此处没有使用#[link]属性指定需要链接的C库， 因为我们在build.rs构建脚本中已经设定好了，
3 //rust cargo 知道该去链接那个C库。
4 extern "C" { fn hello(); }
5
6 fn main() {
7     unsafe { hello(); }
8 }
```

```
1 //src/hello.c
2
3 #include <stdio.h>
4
5 void hello() {
6     printf("Hello, World!\n");
7 }
```

如何自动检测并链接到操作系统中已有的C库，自动检测库名，库版本，库类型等等，自动设定好cargo相关参数，类似Linux pkg-config工具，the pkg-config crate就是对应实现，详情请看：[`https://docs.rs/pkg-config/0.3.17/pkg\\_config/`](https://docs.rs/pkg-config/0.3.17/pkg_config/)，和[`https://doc.rust-lang.org/cargo/reference/build-script-examples.html`](https://doc.rust-lang.org/cargo/reference/build-script-examples.html)

## (5) C call Rust

C 和 Rust互通，需要满足3大原则：

- (1) extern "C" 修饰Rust 函数。
- (2) #[no\_mangle] 修饰Rust函数，使得C Linker认得Rust函数名。
- (3) C and Rust 都认识的数据类型，并且具有相同的内存布局。

---

强调3点：

- (4) C 和Rust互相传递数据对象，因为跨越编程语言边界，所以 必须慎重考虑其回收和释放问题，避免出现“悬指针”或“内存泄露”问题。
- (5) 避免Rust panic跨越边界危及C code，采用[std::panic::catch\\_unwind](#)包装可能发生panic的rust code，从而避免panic蔓延。

Rust语言在设计时就与C互访作为一个重点考虑，比如与C ABI兼容，从而做到二进制互访，以库的形式，最大化利用C语言世界通用语的巨大优势！Rust通吃硬件、嵌入式、操作系统等。

- (6) C和Rust通过回调函数之类互通数据状态，在多线程、异步等并发情况，若访问全局/静态变量时，请慎重考虑“竞态保护”，如锁保护，或是采用rust channel之类读写，以免状态错乱。

**Rust官方推荐使用bindgen/cbindgen工具来自动生产C/Rust兼容代码，因为这两个Rust crate都有Rust官方开发人员加入，可以确保及时准确与Rust更新保持一致！！**

**毕竟Rust非常年轻活跃，进化飞速，所以Rust语言本身及Rust FFI都在不断演化！**

## 【Rust to C 字符串】

Rust type	Intermediate	C type
String	CString	*char
&str	CStr	*const char
()	c_void	void
u32 or u64	c_uint	unsigned int
etc	...	...

详情请看: [https://rust-embedded.github.io/book/interoperability/index.html#interoperability`](https://rust-embedded.github.io/book/interoperability/index.html#interoperability)

## 【build Rust to a c lib】

```

1 #(1) cargo new xxxx --lib
2 #(2) 编辑Cargo.toml
3 [lib]
4 name = "your_crate" #库名, 默认库名就是[package]中定义的名字。
5 crate-type = ["cdylib"] # Creates dynamic lib #与C兼容的动态库。
6 # crate-type = ["staticlib"] # Creates static lib #与C兼容的静态库。
7 #(3) cargo build --release

```

详情请看: [https://rust-embedded.github.io/book/interoperability/rust-with-c.html`](https://rust-embedded.github.io/book/interoperability/rust-with-c.html)

## 【link to rust cdylib/ staticlib from c project】

```

1 #/unsafe_collect/ffi/c2rust/box_t 项目结构
2 #cargo build 在target/debug/中生成libbox.so和libbox.a
3 .
4 |— Cargo.lock
5 |— Cargo.toml
6 |— src
7 |— c_call_rust.c
8 |— lib.rs
9
10 1 directory, 4 files
11

```

```

1 [package]
2 name = "box_t"
3 version = "0.1.0"
4 authors = ["yujinliang <285779289@qq.com>"]
5 edition = "2018"

```



```

6
7 # 定义rust 库名和库类型。
8 [lib]
9 name = "box"
10 crate-type = ["cdylib", "staticlib"]

```

```

1 //src/lib.rs
2 #[repr(C)]
3 #[derive(Debug)]
4 pub struct Foo;
5
6 #[no_mangle]
7 pub extern "C" fn foo_new() -> Box<Foo> {
8     Box::new(Foo)
9 }
10
11 // C `s NULL pointer 对应rust Option::None
12 #[no_mangle]
13 pub extern "C" fn foo_delete(f: Option<Box<Foo>>) {
14     println!("{:?}", f);
15 }

```

```

1 //c_call_rust.c
2 #include <stddef.h>
3 // Returns ownership to the caller.
4 struct Foo* foo_new(void);
5
6 // Takes ownership from the caller; no-op when invoked with NULL.
7 void foo_delete(struct Foo*);
8
9 int main() {
10     foo_delete(foo_new());
11     foo_delete(NULL); //C的空指针NULL 对应为Rust中的Option::None
12 }

```

首先 cargo build 生成C库, 静态库: libbox.a 、动态库: libbox.so

其次动态链接: gcc -o cm src/c\_call\_rust.c -L target/debug/ -lbox

最后运行: LD\_LIBRARY\_PATH=target/debug/ ./cm

详情请看: <http://jakegoulding.com/rust-ffi-omnibus/>

若要gcc静态链接libbox.a, 如下两种方法都可以:

(1)# gcc -o cm src/c\_call\_rust.c -l:libbox.a -L target/debug/ -lpthread -ldl

(2)# gcc -o cm src/c\_call\_rust.c -L target/debug/ -Wl,-Bstatic -lbox -Wl,-Bdynamic -lgcc\_s -ldl -lpthread

注意: -Wl,-Bstatic -l静态库名 , 这几个gcc参数强制要求gcc静态链接静态库。

-Wl,-Bdynamic -l 动态库名, 强制要求gcc动态链接库。注意 “绿色部分参数间不要有空格, 否则无效” ;

-l:静态库全名, 如: -l:libbox.a , 也是要求gcc静态链接这个库。

## 【The bindgen crate】

扫描C/C++ code, 从而自动生成对应的Rust code, 如: 函数声明, 类型定义等, 主攻Rust call C。

```
1 //doggo.h bindgen 扫描 C code。
2 typedef struct Doggo {
3     int many;
4     char wow;
5 } Doggo;
6
7 void eleven_out_of_ten_majestic_af(Doggo* pupper);
8 //-----
9 //doggo.rs
10 //bindgen 自动生成对应的Rust code
11 //从下面生成的Rust code可以看出:Rust call C需要遵循的原则:
12 //(1) 双方都认识的数据类型。 (2) 数据类型的排列, 对齐方式要与C一致, 即相同的内存布局。 (3)
13 函数调用规约要与C一致, 即extern "C" 。
14 //(4) 双方互传的数据对象的回收释放问题要慎重, 避免“悬指针”。
15
16 #[repr(C)]
17 pub struct Doggo {
18     pub many: ::std::os::raw::c_int,
19     pub wow: ::std::os::raw::c_char,
20 }
21
22 extern "C" {
23     pub fn eleven_out_of_ten_majestic_af(pupper: *mut Doggo);
24 }
25
26 //有了C code相应的Rust声明和定义, 再link到指定C库, 就可以调用C函数啦。
```

## 【The cbindgen crate】

扫描Rust code, 从而自动生成对应的C/C++ code, 如: 函数声明, 类型定义等, 主攻于C call Rust。

```
1 //扫描 rust code
2 //repr(C) 和 pub 都要有的类型定义才会被自动生成 C code。
3 #[repr(C)]
4 #[derive(Copy, Clone)]
5 pub struct NumPair {
6     pub first: u64,
7     pub second: usize,
8 }
9
```

```

10 //自动生成C code
11 typedef struct NumPair {
12     uint64_t first;
13     uintptr_t second;
14 } NumPair;

```

```

1 //扫描一个Rust 函数
2 #[no_mangle] //这个属性必须要有，确保C linker认得Rust的函数名。
3 pub extern "C" fn process_pair(pair: NumPair) -> f64 { //pub extern "C" 表明只有公开
    且与C调用规约一致的Rust函数才会被自动生成C code，当然参数类型也要与C匹配才行。
4     (pair.first as f64 * pair.second as f64) + 4.2
5 }
6
7 //自动生成C code
8 double process_pair(NumPair pair);

```

有了Rust code相应的C 声明和定义，再link到指定Rust库，就可以调用Rust函数啦！详情请看：

<https://karroffel.gitlab.io/post/2019-05-15-rust/>，<https://crates.io/crates/bindgen>，<https://crates.io/crates/cbindgen>。对于库的链接方法，Rust和C一样，比如：rustc /gcc -L 库搜索路径 -l 库名 source.c/source.rs...；当然cargo也有相应的配置方法。

Rust 不允许源代码级别和其他编程语言的互访机制！因为代价太大，并且干扰太大！所以Rust只提供二进制库的形式的互访，遵循同样的内存布局和函数调用规约，那么就可以互访！！！相互界限明确，避免互相干扰！！！！

## 【The cpp/cpp\_build crate】

the cpp 和cpp\_build crate 使得直接在Rust 源码中写C/C++ 源码 成为可能！

```

1 //https://github.com/yujinliang/rust_learn/tree/master/unsafe_collect/ffi/rust2c/wri
te_c_in_rust
2
3 use cpp::*;
4
5 cpp!{{
6     #include <stdio.h>
7 }}
8
9 fn add(a: i32, b: i32) -> i32 {
10     unsafe {
11         cpp!([a as "int32_t", b as "int32_t"] -> i32 as "int32_t" {
12             printf("adding %d and %d\n", a, b);
13             return a + b;
14         })
15     }

```

```

16 }
17
18 fn main() {
19     println!("{}", add(1, 7));
20 }

```

详情请看: <https://karroffel.gitlab.io/post/2019-05-15-rust/>, <https://docs.rs/cpp/0.5.4/cpp/>, <https://crates.io/crates/cpp>

## 【Box<T> in FFI, c call rust】

从Rust 1.41.0开始, 我们已经声明了一个`Box<T>`, 其中`T: size`现在与C语言的指针 (`T*`) 类型ABI兼容。因此, 如果有定义了一个`extern "C"` Rust函数, 从C调用, 那么Rust函数的输入参数类型和返回参数类型可以为`Box<T>`, 而相应的C函数声明中对应参数的类型为C语言的`T*`, 强调一下, 这一特性只在C调用Rust的情况下成立, `Box<T>`拥有所有权, 负责管理内存的回收释放, 而C方只是使用, 不关心也不负责其内存的回收和释放! 当然C代码也要遵守规矩, 不允许私自释放`T*`, 也不要超越其生命周期使用`T*`, 如下:

```

1 // C header
2
3 // Returns ownership to the caller.
4 struct Foo* foo_new(void); //此处返回值类型相当于Rust Box<Foo>
5
6 // Takes ownership from the caller; no-op when invoked with NULL.
7 void foo_delete(struct Foo*); // 此处C函数输入参数类型相当于Rust Option<Box<Foo>> .

```

```

1 //对应Rust code
2 #[repr(C)]
3 pub struct Foo;
4
5 #[no_mangle]
6 pub extern "C" fn foo_new() -> Box<Foo> { //此处返回值类型相当于C struct Foo*。
7     Box::new(Foo)
8 }
9
10 // The possibility of NULL is represented with the `Option<_>`.
11 #[no_mangle]
12 pub extern "C" fn foo_delete(_: Option<Box<Foo>>) {} //此处Rust 函数输入参数相当于C struct Foo* 和其为NULL时的情况。

```

再次强调一下, 上面的代码只在C call Rust情况下有效! 但反过来Rust call C时, 函数声明定义在Rust, 而函数实现定义在C, 此时Rust对于C创建的对象没有所有权, 只能使用, 回收和释放都由C掌控! 通俗的将, 谁生的孩子谁养! 双方都遵守这个江湖规矩, 一片祥和! 若是违反大家都完蛋。详情请看: ``

## (6) 自动生成rust code

```
1 #下面是一个rust 工程目录树形结构
2 .
3 |— Cargo.toml
4 |— build.rs
5 |— src
6 |— main.rs
7
8 1 directory, 3 files
```

```
1 #配置Cargo.toml
2 [package]
3 name = "code_generate"
4 version = "0.1.0"
5 authors = ["yujinliang <285779289@qq.com>"]
6 edition = "2018"
7 build="build.rs" # 关键点，启用构建脚本。
8
9 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
10
11 [dependencies]
12
```

```
1 // rust cargo构建脚本: build.rs
2
3 use std::env;
4 use std::fs;
5 use std::path::Path;
6
7 fn main() {
8     //"OUT_DIR" 告诉cargo 此build脚本的output应该存放到什么位置。
9     let out_dir = env::var_os("OUT_DIR").unwrap();
10    let dest_path = Path::new(&out_dir).join("hello.rs");
11    fs::write(
12        &dest_path,
13        "pub fn message() -> &'static str {
14            \"Hello, World!\\n\"
15        }
16        \"
17    ).unwrap();
18
19    //注意哟：这不是普通的print呀， 这是配置cargo的一种官方方法。
20    //“rerun-if-changed”是cargo 指令，下面代码的意思是：只有当build.rs脚本文件发生变化时，
    才重新执行build.rs，
21    //否则默认只要package里的文件发生变化，就re-run build.rs。
22    println!("cargo:rerun-if-changed=build.rs");
```

```
23 }
```

```
1 //src/main.rs
2 //关键点：此行宏代码将build.rs生成的代码文件包含进来加入编译。
3 include!(concat!(env!("OUT_DIR"), "/hello.rs"));
4
5 fn main() {
6     println!("{}", message());
7 }
```

## (7) 条件编译

Rust属性`cfg`、`cfg_attr`和宏`cfg!`是实现条件编译的三剑客，再配合上`build.rs`构建预处理脚本，四驾马车并行不悖，从而实现Rust条件编译。详情请看：<https://doc.rust-lang.org/reference/attributes.html>、<https://doc.rust-lang.org/reference/conditional-compilation.html#the-cfg-attribute>、<https://doc.rust-lang.org/std/macro.cfg.html>、实际的代码例子：<https://github.com/sfackler/rust-openssl/blob/dc72a8e2c429e46c275e528b61a733a66e7877fc/openssl-sys/build/main.rs#L216>

### 【cfg 属性】

```
1 // cfg(predicate), 这个predicate中文意思为：谓词，说白了就是一些判断表达式，最终结果为true/false
2 //而且predicate间可以通过all, any, not 组合起来，表达与、或、非，用于条件组合。
3 //所以下面的函数只在"macos"系统下才会被加入编译。
4 #[cfg(target_os = "macos")]
5 fn macos_only() {
6     // ...
7 }
8
9 //any相当于或的关系，所以只要foo或者bar只要有一个被定义了，则结果为true，即下面的函数就会被加入编译。
10 //提示一下!!! 通常我们在build.rs构建脚本中检测系统环境，从而决定定义foo还是bar。
11 #[cfg(any(foo, bar))]
12 fn needs_foo_or_bar() {
13     // ...
14 }
15
16 //all相当于与的关系，所有条件都为true时，最终结果才为true。所以下面的意思为：首先必须是unix系统并且必须是32bit环境，所有条件都同时满足时下面函数才被加入编译。
17 #[cfg(all(unix, target_pointer_width = "32"))]
18 fn on_32bit_unix() {
19     // ...
20 }
21
22 //not 相当于非的关系，取反的关系。
23 //如果定义了foo，则下面函数不被加入编译，反之加入。
24 #[cfg(not(foo))]
25 fn needs_not_foo() {
```

```
26 // ...
27 }
```

## 【cfg\_attr 属性】

```
1 //例子1
2 #[cfg_attr(linux, path = "linux.rs")] //当linux被预定义时，谓词为真，故此cfg_attr展
开为: #[path = "linux.rs"]
3 #[cfg_attr(windows, path = "windows.rs")] //当windows被预定义时，谓词为真，故此cfg_
attr展开为: #[path = "windows.rs"]
4 mod os;
5 //例子2
6 #[cfg_attr(feature = "magic", sparkles, crackles)] //当谓词: feature = "magic"为真时，
cfg_attr才会展开为如下:
7 #[sparkles]
8 #[crackles]
9 fn bewitched() {}
10
11 //总结: cfg 主攻条件判断，cfg_attr主攻条件满足后自动展开，前者主要在于条件编译，后者主
要在于按条件配置不同属性，两者共同适合那些可以配置属性的rust 元素，如函数，trait, struct,
enum等等，
12 //而宏cfg! 适合用在函数代码逻辑中，if cfg!(predicate) some code else other code，如下
例:
13 let machine_kind = if cfg!(unix) {
14     "unix"
15 } else if cfg!(windows) {
16     "windows"
17 } else {
18     "unknown"
19 };
20
21 println!("I'm running on a {} machine!", machine_kind);
```

## 【cargo feature】

```
1 [package]
2 name = "conditional_compilation"
3 version = "0.1.0"
4 authors = ["yujinliang <285779289@qq.com>"]
5 edition = "2018"
6 build="build.rs"
7 # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/m
anifest.html
8
9
10 [features]
11 default = ["foo_1"] # cargo build /run 默认开启的feature。
12 foo_1 = [] #定义一个feature。
```

```

13 foo_2 = []
14 foo_3 = [] # 方括号中列出此feature依赖的其他feature, 逗号分割。
15
16 #cargo build/run --features "foo_2"
17 #cargo build/run #默认开启default feature
18 #cargo run --features "foo_2 foo_3" #开启编译foo_2 和foo_3 feature。
19
20 [dependencies]
21

```

```

1 //src/main.rs
2 #[cfg(feature = "foo_1")]
3 fn foo_1() {
4     println!("foo_1");
5 }
6
7 #[cfg(feature = "foo_2")]
8 fn foo_2() {
9     println!("foo_2");
10 }
11
12 #[cfg(feature = "foo_3")]
13 fn foo_3() {
14     println!("foo_3");
15 }
16
17 fn foo() {
18     if cfg!(feature = "foo_1") {
19         println!("foo_1");
20     }
21     if cfg!(feature = "foo_2") {
22         println!("foo_2");
23     }
24     if cfg!(feature = "foo_3") {
25         println!("foo_3");
26     }
27 }
28
29 fn main() {
30     foo();
31 }

```

构建编译时，可以选择不同的feature, 从而选择组合不同的功能子集，非常灵活有效，详情请看：  
<https://doc.rust-lang.org/cargo/reference/features.html> 和  
<https://stackoverflow.com/questions/27632660/how-do-i-use-conditional-compilation-with-cfg-and-cargo> , <https://crates.io/crates/cfg-if>



通常我们的工程引用其他creates时，可以指定启用其那些features, Cargo.toml如下：

[dependencies]

serde = {version = "1.0", default-features = false} #不启用默认features

[dev-dependencies]

serde = {version = "1.0", features = ["std"]} #既然是中括号当然是个列表，逗号分割，代表可以启用许多features.

## • 回调函数

C语言的函数指针大家应该都了解，我不再啰嗦！一旦跨越C和Rust的语言边界，大家都遵循C ABI世界语互通，故此回调函数只能采用C语言的函数指针，而Rust闭包实则为Rust语法糖，只有Rust自己认得！

采用Rust闭包当做回调函数非常简洁高效易用，但是C不认识它！为此网上一位高手介绍了一种方法，通过C函数指针、Rust泛型、Rust闭包等，最终间接实现了采用Rust闭包作为回调函数的方法，原文链接：

[`http://adventures.michaelfbryan.com/posts/rust-closures-in-ffi/`](http://adventures.michaelfbryan.com/posts/rust-closures-in-ffi/)，有兴趣大家可以看看，我自己也参照其原文整理出了两个代码例子，代码地址：

[`https://github.com/yujinliang/rust\\_learn/tree/master/unsafe\\_collect/ffi/c2rust/closure\\_as\\_callback`](https://github.com/yujinliang/rust_learn/tree/master/unsafe_collect/ffi/c2rust/closure_as_callback)，

[`https://github.com/yujinliang/rust\\_learn/tree/master/unsafe\\_collect/ffi/c2rust/simple`](https://github.com/yujinliang/rust_learn/tree/master/unsafe_collect/ffi/c2rust/simple)

## • 交叉编译

这是一个复杂庞大的主题，我不主攻这方面，所以不再班门弄斧，收集一些好资料简单学习一下，以后有精力再深入学习，Rust 交叉编译好资料：<https://github.com/japaric/rust-cross>，<https://os.phil-opp.com/cross-compile-libcore/>，交叉编译工具：<https://github.com/rust-embedded/cross>，<https://github.com/japaric/xargo>，<https://forge.rust-lang.org/release/platform-support.html>，<https://os.phil-opp.com/freestanding-rust-binary/>

嵌入式开发分为两大类，如下：

【Bare Metal Environments裸机硬件】

没有操作系统，程序直接运行在裸机中，`#![no_std]` is a crate-level attribute that indicates that the crate will link to the core-crate instead of the std-crate.

[`https://doc.rust-lang.org/core/`](https://doc.rust-lang.org/core/)，[`https://doc.rust-lang.org/std/`](https://doc.rust-lang.org/std/)，说白了不能使用rust std, 只能使用其微缩版rust-libcore，一个no\_std小例子：<https://docs.rust-embedded.org/embedonomicon/smallest-no-std.html>，

【Hosted Environments有操作系统】

硬件上有操作系统，如：linux /windows/macos/ios，rust 程序可以引用rust std，可以理解为我们通常使用的PC环境，或者可以运行操作系统的环境。

## • 关于作者：

作者：心尘了

email: [285779289@qq.com](mailto:285779289@qq.com)

git: <https://github.com/yujinliang>

CSDN ID: htyu\_0203\_39

知乎: <https://www.zhihu.com/people/xin-chen-liao>

智商尚可，情商归零，早年入行，不善文章，埋头码农，而今不惑之年，多年风雨，恍然而过，志在何方？！  
我斗胆喊他几嗓子，好不好，对不对，您就权当一乐！  
我就是想告诉这世界，我来过！

- **参考资料：**

<https://stackoverflow.com/questions/24145823/how-do-i-convert-a-c-string-into-a-rust-string-and-back-via-ffi>

<https://doc.rust-lang.org/nomicon/ffi.html>

<https://michael-f-bryan.github.io/rust-ffi-guide/>

<http://jakegoulding.com/rust-ffi-omnibus/>

<https://rust-embedded.github.io/book/interoperability/c-with-rust.html>

<https://dev.to/verkkokauppa/creating-an-ffi-compatible-c-abi-library-in-rust-5dji>

<https://doc.rust-lang.org/std/ffi/>

<https://thefullsnack.com/en/string-ffi-rust.html>

<https://github.com/alexcrichon/rust-ffi-examples>

<http://adventures.michaelfbryan.com/posts/rust-closures-in-ffi/>

<https://github.com/Michael-F-Bryan/rust-closures-and-ffi>

<https://github.com/mystor/rust-cpp>

<https://www.cs-fundamentals.com/c-programming/static-and-dynamic-linking-in-c>

<https://stackoverflow.com/questions/40833078/how-do-i-specify-the-linker-path-in-rust>

<https://doc.rust-lang.org/cargo/reference/build-script-examples.html>

<https://doc.rust-lang.org/reference/conditional-compilation.html#the-cfg-attribute>

<https://doc.rust-lang.org/std/macro.cfg.html>

<https://github.com/sfackler/rust->

[openssl/blob/dc72a8e2c429e46c275e528b61a733a66e7877fc/openssl-sys/build/main.rs#L216](https://github.com/sfackler/rust-openssl/blob/dc72a8e2c429e46c275e528b61a733a66e7877fc/openssl-sys/build/main.rs#L216)

<https://doc.rust-lang.org/reference/attributes.html>

<https://doc.rust-lang.org/cargo/reference/build-scripts.html#rustc-cfg>

<https://stackoverflow.com/questions/27632660/how-do-i-use-conditional-compilation-with-cfg-and-cargo>

<https://doc.rust-lang.org/cargo/reference/features.html>

<https://github.com/rust-lang/rust-bindgen>

<https://github.com/eqrion/cbindgen>

<https://karroffel.gitlab.io/post/2019-05-15-rust/>

《深入浅出Rust》范长春著，机械工业出版社

<https://crates.io/crates/cpp>

[https://crates.io/crates/cpp\\_build](https://crates.io/crates/cpp_build)

<https://blog.rust-lang.org/2020/05/15/five-years-of-rust.html>

<https://blog.rust-lang.org/2020/01/30/Rust-1.41.0.html>

<https://rust-embedded.github.io/book/interoperability/rust-with-c.html>

<https://www.zhihu.com/question/22940048>

<https://stackoverflow.com/questions/43826572/where-should-i-place-a-static-library-so-i-can-link-it-with-a-rust-program>

[https://doc.rust-lang.org/std/panic/fn.catch\\_unwind.html](https://doc.rust-lang.org/std/panic/fn.catch_unwind.html)

<https://www.worthe-it.co.za/programming/2018/11/18/compile-time-feature-flags-in-rust.html>

<https://stackoverflow.com/questions/37526598/how-can-i-override-a-constant-via-a-compiler-option>

<https://doc.rust-lang.org/cargo/reference/manifest.html>

<https://doc.rust-lang.org/reference/conditional-compilation.html>

<https://github.com/japaric/rust-cross>

<https://forge.rust-lang.org/compiler/cross-compilation/windows.html>

<https://github.com/rust-embedded/cross>

<https://os.phil-opp.com/cross-compile-libcore/>

<https://rust-embedded.github.io/book/intro/index.html>

<https://github.com/japaric/xargo>

<https://docs.rust-embedded.org/embedonomicon/smallest-no-std.html>

<https://forge.rust-lang.org/release/platform-support.html>

<https://os.phil-opp.com/freestanding-rust-binary/>

<http://www.aosabook.org/en/llvm.html>

<http://jonathan2251.github.io/lbd/about.html>

<https://jonathan2251.github.io/lbd/backendstructure.html>

<https://github.com/ilo5u/CX-CPU>

<https://repository.iiitd.edu.in/jspui/bitstream/handle/123456789/345/MT13011.pdf;sequence=1>

<http://jonathan2251.github.io/lbt/index.html>

<https://www.llvm.org/docs/WritingAnLLVMBackend.html>

<https://github.com/c64scene-ar/llvm-6502>

<https://elinux.org/images/b/b7/LLVM-ELC2009.pdf>

<https://github.com/earl1k/llvm-z80>

<https://github.com/openrisc/llvm-or1k>

<http://jonathan2251.github.io/lbd/lbdContents.pdf>

