Rust

引用

可变引用常见错误

```
//可变引用与可变变量产生数据竞争，导致编译失败。
//error[E0499]: cannot borrow `a` as mutable more than once at a time
fn main() {
    let mut a = String::from("abc");
    let b = &mut a;
    a.push_str("ghi");
    b.push_str("def");
    println!("{}", a);
}

```

&和ref

> 解引用通过实现trait Deref

```
fn main() {
    let ref a = 100_i32;
    //等于
    let a = &100_i32;
}

```

字符串

```
let mut s2: String = String::from(s);
s.as_bytes();
s.bytes();
s.chars();
s.chars().count();
s.chars();
s.contains("Hello");
s2.clear();
s.parse::<i32>().unwrap();
s.trim();

```

向量

```
let mut vect: Vec<i32> = vec![100, 200, 300, 400, 500];
vect.get(0);
vect[0];
vect[0] = 120;
vect.push(600);
```

```rust
vect.insert(2, 250);
vect.remove(0);
vect.append(&mut vec![700, 800]);
//vect一般是引用（&vect或&mut vect）
for i in &vect {
    println!("{i}");
}
vect.pop();
vect.clear();

```

哈希表

```rust
let mut hm: HashMap<String, i32> = HashMap::new();
hm.insert(String::from("k"), 100);
hm.get("k").copied().unwrap();
hm.entry(String::from("k")).or_insert(100);
hm.remove("k");
hm.remove_entry("k").unwrap();

```

迭代器

```rust
let iter: Copied<Iter<'_, i32>> = vect.iter().copied();
iter.clone().filter(|x| *x == 300);
iter.clone().map(|x| x * x);
iter.clone().collect::<Vec<i32>>();
iter.clone().chain(vec![600, 700, 800].iter().copied());
iter.clone().enumerate();
iter.clone().count();

```

Result/Option处理

```rust
let err: Result<String, Error> = read_to_string("a.txt");
err.unwrap();
err.expect("···");
err.unwrap_or(String::from("Error!"));
err.unwrap_or_default();
err.unwrap_or_else(|e| format!("Error:{}", e.to_string()));
err?;
//Result特有
err.is_ok();
err.is_err();
//Option特有
opt.is_some();
opt.is_none();

```

Derive

```rust

```

```rust
/// Debug：通过实现 Debug trait，可以使用 println!("{:?}", my_struct) 来打印结构体的调试信息。
/// Copy：可以按位复制（不含引用等），可以在赋值时自动复制，所有实现了 Copy 的类型都必须实现 Clone。
/// Clone：通过实现 Clone trait，可以使用 my_struct.clone() 创建结构体的克隆副本。
/// PartialEq 和 Eq：通过实现 PartialEq trait，可以进行结构体的部分相等性比较，而 Eq trait 则实现了完全相等性比较。
///    PartialEq实现等于和不等于判断，Eq没有方法，但只要实现（必须也实现PartialEq），就满足：
///        1.a == a；
///        2.如果a == b，那么b == a；
///        3.如果a == b且b == c，那么a == c。
/// PartialOrd 和 Ord：通过实现 PartialOrd trait，可以对结构体进行部分有序性比较，而 Ord trait 实现了完全有序性比较。
#[derive(Debug,Clone)]
struct MyStruct {
    // 结构体字段
}

```

类型转换

```rust
let decimal = 65.4321_f32;
let integer = decimal as u8;
let character = integer as char;
```

类型别名

```rust
type NanoSecond = u64;
//泛型别名
type AliasedResult<T> = Result<T, ParseIntError>;
```

运算符重载

```rust
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;

// `std::ops::Add` trait 用来指明 `+` 的功能，这里我们实现 `Add<Bar>`，它是用于
// 把对象和 `Bar` 类型的右操作数（RHS）加起来的 `trait`。
// 下面的代码块实现了 `Foo + Bar = FooBar` 这样的运算。
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");
```

```
        FooBar
    }
}

// 通过颠倒类型，我们实现了不服从交换律的加法。
// 这里我们实现 `Add<Foo>`，它是用于把对象和 `Foo` 类型的右操作数加起来的 trait。
// 下面的代码块实现了 `Bar + Foo = BarFoo` 这样的运算。
impl ops::Add<Foo> for Bar {
    type Output = BarFoo;

    fn add(self, _rhs: Foo) -> BarFoo {
        println!("> Bar.add(Foo) was called");

        BarFoo
    }
}

fn main() {
    println!("Foo + Bar = {:?}", Foo + Bar);
    println!("Bar + Foo = {:?}", Bar + Foo);
}

```

自动推断类型

```
//常用于泛型参数中：
fn main() {
    let x: Vec<_> = (0..10).collect();
}

```

常量

```
///const：不可改变的值（通常使用这种）。
///static：具有 'static 生命周期的，可以是可变的变量（static mut）。
///有个特例就是 "string" 字面量。它可以不经改动就被赋给一个 static 变量，因为它的类型标记：&'static str 就包含了所要求的生命周期 'static。其他的引用类型都必须特地声明，使之拥有 'static 生命周期。这两种引用类型的差异似乎也无关紧要，因为无论如何，static 变量都得显式地声明。
static A: &'static str = "Rust";
static mut B: &'static str = "Rust";
const C: &str = "Rust";
```

From与Into

> From trait 允许一种类型定义 “怎么根据另一种类型生成自己”，因此它提供了一种类型转换的简单机制。
> Into trait 就是把 From trait 倒过来而已。也就是说，如果你为你的类型实现了 From，那么同时你也就免费获得了 Into。

```
use std::convert::From;

```

```rust
#[derive(Debug)]
struct Number {
    value: i32,
}

impl From<i32> for Number {
    fn from(item: i32) -> Self {
        Number { value: item }
    }
}

fn main() {
    let num = Number::from(30);
    let num: Number = 30.into();
    println!("My number is {:?}", num);
}

```

> `TryFrom` 和 `TryInto` trait 用于易出错的转换，也正因如此，其返回值是 `Result` 型。

> 要把任何类型转换成 `String`，您应该实现 `fmt::Display` trait，它会自动提供 `ToString`，并且还可以用来打印类型，就像 `print!` 一节中讨论的那样。See:ToString、fmt::Display, FromStr

文件分层

```
.
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs

```

https://rustwiki.org/zh-CN/rust-by-example/mod/split.html

Drop

https://rustwiki.org/zh-CN/rust-by-example/trait/drop.html

```rust
struct Droppable {
    name: &'static str,
}

// 这个简单的 `drop` 实现添加了打印到控制台的功能。
impl Drop for Droppable {
    fn drop(&mut self) {
        println!("> Dropping {}", self.name);
    }
}

fn main() {
    let _a = Droppable { name: "a" };

    // 代码块 A
    {
        let _b = Droppable { name: "b" };

        // 代码块 B
        {
            let _c = Droppable { name: "c" };
            let _d = Droppable { name: "d" };

            println!("Exiting block B");
        }
        println!("Just exited block B");

        println!("Exiting block A");
    }
    println!("Just exited block A");

    // 变量可以手动使用 `drop` 函数来销毁。
    drop(_a);
    //或std::mem::drop(_a);

    println!("end of the main function");

    // `_a` 不会在这里再次销毁，因为它已经被（手动）销毁。
}

```

Crate

https://rustwiki.org/zh-CN/rust-by-example/crates.html

```
# 默认情况下，库会使用 crate 文件的名字，前面加上 “lib” 前缀，但这个默认名称可以使用 crate_name 属性 覆盖。
rustc --crate-type=lib rary.rs
rustc main.rs --extern rary=library.rlib --edition=2018
```

```
[package]
# See: https://doc.rust-lang.org/cargo/reference/manifest.html
name = "hello_world"
version = "0.1.0"
edition = "2018"
sql = "0.4.3"# <-- Add dependency here
```

cfg

`#[cfg()]` 是 Rust 中的一个属性，用于根据配置条件来选择性地包含或排除代码。

`#[cfg()]` 属性中使用的逻辑运算符有以下几种：

1. `all(expr1, expr2, ...)`：逻辑与运算符，要求所有条件表达式都为真才返回真。例如：`#[cfg(all(feature = "foo", target_os = "linux"))]` 表示只有在启用 "foo" 功能并且目标操作系统是 Linux 时条件成立。
2. `any(expr1, expr2, ...)`：逻辑或运算符，只要有任一条件表达式为真就返回真。例如：`#[cfg(any(feature = "foo", feature = "bar"))]` 表示只要启用 "foo" 或 "bar" 中的任意一个功能时条件成立。
3. `not(expr)`：逻辑非运算符，对条件表达式取反。例如：`#[cfg(not(debug_assertions))]` 表示只有在非调试断言模式下条件成立。

```
#[cfg(target_os = "linux")]
fn only_on_linux() {
    // 仅在 Linux 系统上编译和执行的代码
}

#[cfg(all(unix, not(target_os = "linux")))]
fn on_unix_but_not_linux() {
    // 仅在 Unix 系统但不是 Linux 上编译和执行的代码
}

#[cfg(any(windows, target_os = "macos"))]
fn on_windows_or_macos() {
    // 仅在 Windows 或 macOS 上编译和执行的代码
}

#[cfg(not(debug_assertions))]
fn when_not_debug_assertions() {
    // 仅在非调试断言模式下编译和执行的代码
}

#[cfg(feature = "my_feature")]
fn with_my_feature_enabled() {
    // 仅在启用 "my_feature" 功能时编译和执行的代码
}

```

```
if cfg!(target_os = "linux") {
    println!("Yes. It's definitely linux!");
} else {
    println!("Yes. It's definitely *not* linux!");
}

```

宏

```
macro_rules! $name {
    $rule0 ;
    $rule1 ;
    // …
    $ruleN ;
}
```

$rule:

`($pattern:$identifier) => {$expansion}`

$identifier:

- `item:` 条目，比如函数、结构体、模组等。
- `block:` 区块(即由花括号包起的一些语句加上/或是一项表达式)。
- `stmt:` 语句
- `pat:` 模式
- `expr:` 表达式
- `ty:` 类型
- `ident:` 标识符
- `path:` 路径 (例如 `foo, ::std::mem::replace, transmute::<_, int>, …`)
- `meta:` 元条目，即被包含在 `#[...]`及`#![...]`属性内的东西。
- `tt:` 标记树

重复

- `$` 是字面标记。
- `( ... )` 代表了将要被重复匹配的模式，由小括号包围。
- `sep`是一个可选的分隔标记。常用例子包括`,`和`;`。
- `rep`是重复控制标记。当前有两种选择，分别是`*` (代表接受0或多次重复)以及`+` (代表1或多次重复)。目前没有办法指定“0或1”或者任何其它更加具体的重复计数或区间。

```
macro_rules! vec_strs {
    (
        // 重复开始：
        $(
            // 每次重复必须有一个表达式...
            $element:expr
        )
        // ...重复之间由“,”分隔...
        ,
        // ...总共重复0或多次.
        *
    ) => {
        // 为了能包含多条语句，
        // 我们将扩展部分包裹在花括号中...
        {
            let mut v = Vec::new();
            // 重复开始：
            $(
                // 每次重复将包含如下元素，其中
```

```
                // “$element”将被替换成其相应的展开...
                v.push(format!("{}", $element));
            )*
            v
        }
    };
}

```

Mutex和Arc

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}

```

智能指针

- `Box<T>`：它提供了最简单的堆资源分配方式，单所有权。Box类型拥有其中的值，并且可用于保存结构体中的值，或者从函数返回它们。
- `Rc<T>`：它用于引用计数。每当获取新引用时，计数器会执行递增操作，并在用户释放引用时对计数器执行递减操作。当计数器的值为零时，该值将被移除。
- `Arc<T>`：它用于原子引用计数。这与之前的类型类似，但具有原子性以保证多线程的安全性。
- `Cell<T>`：它为我们提供实现了Copy特征的类型的内部可变性。换句话说，我们有可能获得多个可变引用。
- `RefCell<T>`：它为我们提供了类型的内部可变性，并且不需要实现Copy特征。它用于运行时的锁定以确保安全性。

RefCell内部可变性

```rust
let a = RefCell::new(100_i32);
//编译失败（RefCell不能解引用）：
// println!("{}", *a);
println!("{}", *a.borrow());
//编译失败（a不可变）：
// *a = 120;
*a.borrow_mut() = 120;
//Panic（同时两个可变引用）：
let b = a.borrow_mut();
let c = a.borrow_mut();
```

RefCell+Rc: 多个所有者 并且 可以修改值

```rust
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));
    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
    let b = Cons(Rc::clone(&value), Rc::clone(&a));
    *value.borrow_mut() += 10;
    println!("{:?}\n{:?}", a, b);
}

```

强/弱引用

```rust
let strong = Rc::new(100_i32);
//由强引用创建弱引用。
let weak = Rc::downgrade(&strong);
//由弱引用创建强引用。
//如果弱引用已无效，就返回None。
let strong = Weak::upgrade(&weak).unwrap();

```

编译

编译模式

```
cargo build
cargo build --release

```

编译目标类型

链接(linkage) - Rust 参考手册 中文版 (rustwiki.org)

- `--crate-type=bin` 或 `#[crate_type = "bin"]` - 将生成一个可执行文件。这就要求在 crate 中有一个 `main` 函数，它将在程序开始执行时运行。这将链接所有 Rust 和本地依赖，生成一个单独的可分发的二进制文件。此类型为默认的 crate 类型。
- `--crate-type=lib` 或 `#[crate_type = "lib"]` - 将生成一个 Rust库(library)。但最终会确切输出/生成什么类型的库在未生成之前还不好清晰确定，因为库有多种表现形式。使用 `lib` 这个通用选项的目的是生成“编译器推荐”的类型的库。像种指定输出库类型的选项在 rustc 里始终可用，但是每次实际输出的库的类型可能会随着实际情况的不同而不同。其它的输出（库的）类型选项都指定了不同风格的库类型，而 `lib` 可以看作是那些类型中的某个类型的别名（具体实际的输出的类型是编译器决定的）。
- `--crate-type=dylib` 或 `#[crate_type = "dylib"]` - 将生成一个动态 Rust库。这与 `lib` 选项的输出类型不同，因为这个选项会强制生成动态库。生成的动态库可以用作其他库和/或可执行文件的依赖。这种输出类型将创建依赖于具体平台的库（Linux 上为 `*.so`，macOS 上为 `*.dylib`、Windows 上为 `*.dll`）。
- `--crate-type=staticlib` 或 `#[crate_type = "staticlib"]` - 将生成一个静态系统库。这个选项与其他选项的库输出的不同之处在于——当前编译器永远不会尝试去链接此 `staticlib` 输出[1]。此选项的目的是创建一个包含所有本地 crate 的代码以及所有上游依赖的静态库。此输出类型将在 Linux、macOS 和 Windows(MinGW) 平台上创建 `*.a` 归档文件(archive)，或者在 Windows(MSVC) 平台上创建 `*.lib` 库文件。在这些情况下，例如将 Rust代码链接到现有的非 Rust应用程序中，推荐使用这种类型，因为它不会动态依赖于其他 Rust 代码。
- `--crate-type=cdylib` 或 `#[crate_type = "cdylib"]` - 将生成一个动态系统库。如果编译输出的动态库要被另一种语言加载使用，请使用这种编译选项。这种选项的输出将在 Linux 上创建 `*.so` 文件，在 macOS 上创建 `*.dylib` 文件，在 Windows 上创建 `*.dll` 文件。
- `--crate-type=rlib` 或 `#[crate_type = "rlib"]` - 将生成一个“Rust库”。它被用作一个中间构件，可以被认为是一个“静态 Rust库”。与 `staticlib` 类型的库文件不同，这些 `rlib` 类型的库文件以后会作为其他 Rust代码文件的上游依赖，未来对那些 Rust代码文件进行编译时，那时的编译器会链并解释此 `rlib` 文件。这本质上意味着（那时的） `rustc` 将在（此） `rlib` 文件中查找元数据(metadata)，就像在动态库中查找元数据一样。跟 `staticlib` 输出类型类似，这种类型的输出常配合用于生成静态链接的可执行文件(statically linked executable)。
- `--crate-type=proc-macro` 或 `#[crate_type = "proc-macro"]` - 生成的输出类型没有被指定，但是如果通过 `-L` 提供了路径参数，编译器将把输出构件识别为宏，输出的宏可以被其他 Rust 程序加载使用。使用此 crate 类型编译的 crate 只能导出过程宏。编译器将自动设置 `proc_macro` 属性配置选项。编译 crate 的目标平台(target)总是和当前编译器所在平台一致。例如，如果在 `x86_64` CPU 的 Linux 平台上执行编译，那么目标将是 `x86_64-unknown-linux-gnu`，即使该 crate 是另一个不同编译目标的 crate 的依赖。

Cargo.toml

字段含义

> See The Manifest Format - The Cargo Book (rust-lang.org)

> [[xxx]]双中括号表示可以有多个

- `cargo-features` — 不稳定的、仅限夜间的功能。
- `[package]` — 定义包。
  - `name` — 包的名称。
  - `version` — 包的版本。
  - `authors` — 包的作者。
  - `edition` — Rust 版本。
  - `rust-version` — 支持的最低 Rust 版本。
  - `description` — 包的描述。

    - `documentation` — 软件包文档的 URL。
    - `readme` — 软件包的 README 文件的路径。
    - `homepage` — 软件包主页的 URL。
    - `repository` — 包源存储库的 URL。
    - `license` — 软件包许可证。
    - `license-file` — 许可证文本的路径。
    - `keywords` — 包的关键字。
    - `categories` — 包的类别。
    - `workspace` — 包的工作区路径。
    - `build` — 包构建脚本的路径。
    - `links` — 包链接的本机库的名称。
    - `exclude` — 发布时要排除的文件。
    - `include` — 发布时要包含的文件。
    - `publish` — 可用于阻止发布包。
    - `metadata` — 外部工具的额外设置。
    - `default-run` — 由 `cargo run` 运行的默认二进制文件。
    - `autobins` — 禁用二进制自动发现。
    - `autoexamples` — 禁用示例自动发现。
    - `autotests` — 禁用测试自动发现。
    - `autobenches` — 禁用工作台自动发现。
    - `resolver` — 设置要使用的依赖关系解析程序。
- 目标表：（有关设置，请参阅配置）
    - `[lib]` — 库目标设置。
    - [`[[bin]]`](https://doc.rust-lang.org/cargo/reference/cargo-targets.html#binaries) — 二进制目标设置。
    - [`[[example]]`](https://doc.rust-lang.org/cargo/reference/cargo-targets.html#examples) — 目标设置示例。
    - [`[[test]]`](https://doc.rust-lang.org/cargo/reference/cargo-targets.html#tests) — 测试目标设置。
    - [`[[bench]]`](https://doc.rust-lang.org/cargo/reference/cargo-targets.html#benchmarks) — 基准目标设置。
- 依赖项表：
    - `[dependencies]` — 包库依赖项。
    - `[dev-dependencies]` — 示例、测试和基准测试的依赖项。
    - `[build-dependencies]` — 构建脚本的依赖关系。
    - `[target]` — 特定于平台的依赖项。
- `[badges]` — 要在注册表中显示的锁屏提醒。
- `[features]` — 条件编译功能。
- `[lints]` — 为此软件包配置 linter。
- `[patch]` — 覆盖依赖项。
- `[replace]` — 覆盖依赖项（已弃用）。
- `[profile]` — 编译器设置和优化。
- `[workspace]` — 工作区定义。

默认

```toml
[package]
name = "tool_name"
version = "0.1.0"
edition = "2021"

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

[dependencies]

```

常用

```toml
[package]
name = "hello_opensource"
version = "0.1.0"
authors = ["user<user@mail.com>"]
edition = "2018"

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

[dependencies]
rand = "0.3.14"

```

更新依赖

```
cargo update [-p rand]
```

描述

```toml
[package]
#-snip-
description = "A fun game where you guess what number the computer has chosen."
```

license

```toml
[package]
#-snip-
license = "MIT"
```

bin

```toml
[[bin]]
#-snip-
name = "tool_bin_name"# 可执行文件名源，会查找src/bin/tool_bin_name.rs或src/bin/tool_bin_name/main.rs
test = false
bench = false
```

lib

```toml
[lib]
name = "foo"           # 目标名
path = "src/lib.rs"    # 目标的源文件
crate-type = ["lib"]   # 要生成的crate类型，可选的有bin, lib, rlib, dylib, cdylib, staticlib, and proc-macro
test = true            # 默认为已测试
doc = true             # 是否包含在默认情况下，由Cargo Doc生成文档
edition = "2015"       # 定义目标将使用的 Rust 版本，通常不应设置此字段
bench = true           # Is benchmarked by default.
plugin = false         # Used as a compiler plugin (deprecated).
proc-macro = false     # Set to `true` for a proc-macro library.
harness = true         # Use libtest harness.
required-features = [] # Features required to build this target (N/A for lib).

```

优化等级

```toml
#opt-level 设置控制 Rust 会对代码进行何种程度的优化。这个配置的值从 0 到 3。越高的优化级别需要更多的时间编译，所以如果你在进行开发并经常编译，可能会希望在牺牲一些代码性能的情况下减少优化以便编译得快一些。因此 dev 的 opt-level 默认为 0。当你准备发布时，花费更多时间在编译上则更好。只需要在发布模式编译一次，而编译出来的程序则会运行很多次，所以发布模式用更长的编译时间换取运行更快的代码。这正是为什么 release 配置的 opt-level 默认为 3。
[profile.dev]
opt-level = 0

[profile.release]
opt-level = 3

```

去除debug信息

```toml
[profile.release]
# 可选"none", "debuginfo"和"symbols",
# 或者是"true"和"false",
# 默认strip = "none"
strip = "debuginfo"

```

工作空间

> See Cargo 工作空间 - Rust 程序设计语言 简体中文版 (kaisery.github.io)

汇编ASM

> See 内联汇编 - Rust语言圣经(Rust Course)

```rust
use std::arch::asm;

fn main() {
    let mut x: u64 = 3;
    unsafe {
        //out表示输出，in表示输入，inout表示既输入又输出（可以保证使用同一个寄存器来完成任务）。
        //reg表示让编译器自动选择合适的寄存器。
        //可以像格式化字符串一样用{0}等复用。
        asm!("add {0}, 5", inout(reg) x);
    }
    println!("{}", x);
}

```