Effective Rust 与开发技巧

计 16 陈英豪

2024 年 7 月 17 日

《Effective Rust》

"35 Specific Ways to Improve Your Rust Code"

旨在帮助你写出更**高效，简洁，易维护**的代码

- 将方法的内容、原因与例子一起写出
- 着重对许多 Rust 特有的概念和方法的解读

https://www.lurklurk.org/effective-rust/

课程目的

- 导读 《Effective Rust》，介绍部分代码编写技巧和 Rust 的特有功能
- 成为一份 Reference，帮助各位在写类似 OJ 项目的过程中提高代码质量
- 了解学习 Rust 的意义和价值


课程内容

第一部分

- 构造类型系统（Item 1）
- 使用函数、Trait 来表达代码的行为（Item 2，Item 12）
- Option、Result 及相关的一系列成员函数（Item 3，Item 18）
- Result 与错误处理（Item 4）
- 迭代器及其相关函数（Item 9）

第二部分

- 在未来 Rust 可以帮到你的课程

构建类型系统

什么是类型系统

用于描述项目所要解决的问题，是构建项目的第一步

```rust
#[derive(Deserialize, Serialize, Debug, Clone)]
#[serde(deny_unknown_fields)]
pub struct Problem {
    pub id: u64,
    name: String,
    #[serde(rename = "type")]
    ty: ProblemType,
    #[serde(default)]
    misc: Option<Message>,
    cases: Vec<Case>,
}
```

```rust
#[derive(Deserialize, Serialize, Debug, Clone)]
enum ProblemType {
    #[serde(rename = "standard")]
    Standard,
    #[serde(rename = "strict")]
    Strict,
    #[serde(rename = "spj")]
    Spj,
    #[serde(rename = "dynamic_ranking")]
    DynamicRanking,
}
```

```rust
#[derive(Deserialize, Serialize, Debug, Clone, Default)]
#[serde(deny_unknown_fields)]
pub struct Case {
    pub score: f64,
    pub input_file: PathBuf,
    pub answer_file: PathBuf,
    pub time_limit: u64,
    pub memory_limit: u64,
    #[serde(default)]
    pub shortest_time: u64
}
```

数据类型是有结构的

enum 允许同一个类型有多种互斥的可能形式，可以看作对类型的“或”操作、“+”操作

struct 则允许在同一个类型中组合多种不同类型，可以看作对类型的“与”操作、“x”操作

借助它们，可以将程序的一些不变量直接编码进你所构建的类型系统中

优点：

表达力强：可以清晰地表达复杂的数据结构，定义明确的类型。
类型安全：编译器可以检查数据类型的使用，减少运行时错误。
模式匹配：许多函数式编程语言支持模式匹配，便于对不同类型的值进行处理。

```rust
pub enum VarType {
    I32,
    F32,
    I32Ptr,
    F32Ptr,
    Void,
}
```

```rust
pub enum Value {
    Int(i32),
    Float(f32),
    Temp(LlvmTemp),
}
```

```rust
pub struct BasicBlock<T: InstrTrait<U>, U: TempTrait> {
    pub id: i32,
    pub weight: f64,
    pub kill_size: i32,
    pub prev: Vec<Node<T, U>>,
    pub succ: Vec<Node<T, U>>,
    pub defs: HashSet<U>,
    pub uses: HashSet<U>,
    pub kills: HashSet<U>,
    pub live_in: HashSet<U>,
    pub live_out: HashSet<U>,
    pub phi_instrs: Vec<PhiInstr>,
    pub phi_defs: HashSet<LlvmTemp>,
    pub instrs: Vec<T>,
    pub jump_instr: Option<T>,
}
```

表达力强

在还没有完成项目时，就可以确定好项目的框架，并能够按部就班地补全、丰富项目内容

```rust
struct Config {
    server: Server,
    problems: Vec<Problem>,
    languages: Vec<Language>,
}
```

```rust
struct Server {
    bind_address: String,
    bind_port: u64,
}
```

```rust
pub struct Problem {
    pub id: u64,
    name: String,
    #[serde(rename = "type")]
    ty: ProblemType,
    #[serde(default)]
    misc: Option<Message>,
    cases: Vec<Case>,
}
```

```rust
enum ProblemType {
    #[serde(rename = "standard")]
    Standard,
    #[serde(rename = "strict")]
    Strict,
    #[serde(rename = "spj")]
    Spj,
    #[serde(rename = "dynamic_ranking")]
    DynamicRanking,
}
```

类型安全

将所有可能的情形编码在类型系统中，借助编译器的类型检查直接消除不合法情况

```rust
pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub monochrome: bool,
    // `fg_color` must be (0, 0, 0) if `monochrome` is true.
    pub fg_color: RgbColor,
}
```

```rust
pub enum Color {
    Monochrome,
    Foreground(RgbColor),
}

pub struct DisplayProps {
    pub x: u32,
    pub y: u32,
    pub color: Color,
}
```

模式匹配

Rust 为了方便用户构造类型系统，提供了强大的模式匹配功能

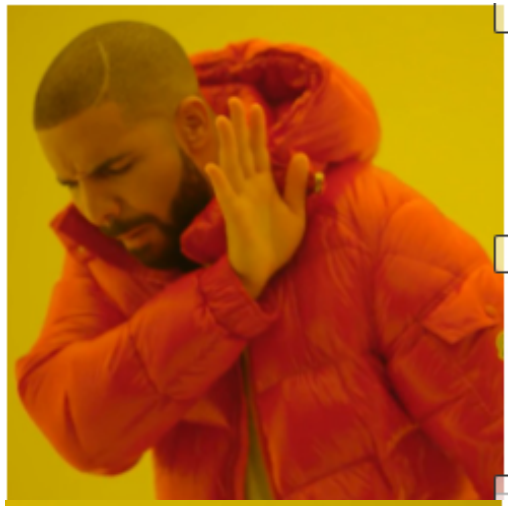
如 if let, while let, match

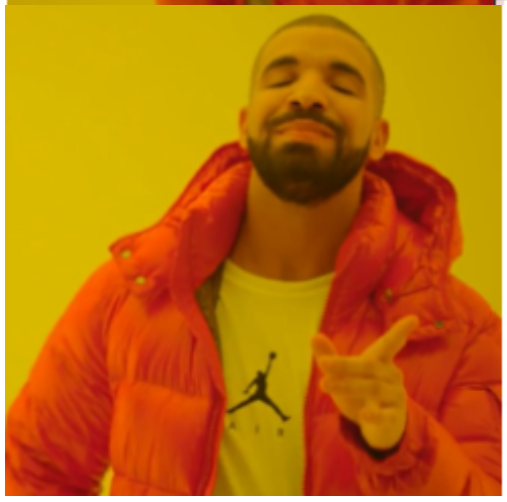
善用它们，以规避非法情况，减少 bug，同时提高可读性和可维护性

```rust
pub enum Value {
    Int(i32),
    Float(f32),
    Temp(LlvmTemp),
}

impl Eq for Value {}

impl Hash for Value {
    fn hash<H: Hasher>(&self, state: &mut H) {
        core::mem::discriminant(self).hash(state);
        match self {
            Value::Int(i: &i32) => {
                i.hash(state);
            }
            Value::Float(f: &f32) => {
                let mut value: f32 = *f;
                if value.is_nan() || value.is_infinite() {
                    value = 1926.0817f32;
                }
                value.to_bits().hash(state);
            }
            Value::Temp(t: &LlvmTemp) => {
                t.hash(state);
            }        Lethe Lee, 6 months ago • feat: common su
        }
    }
}
```

使用函数、Trait 来表达代码的行为

使用函数来表达代码的行为





你是否有过如下经历：

1. 对大段代码进行 Ctrl-C Ctrl-V
2. 编写超过500行的“函数”
3. 自己曾经写下的函数现在却看不懂了

你需要：

1. 保持函数简洁：每个函数只做一件事，明确输入输出和函数的语义（借助类型系统）
2. 文档和注释：提供清晰的文档和注释。

（当你发现你在复制粘贴、函数超过了500行、看不懂自己写的函数时，你就该思考是不是应该拆分函数了）

弱可读性的例子

```
struct Block {
   id: usize,
}

let mut id_to_blocks: HashMap<usize, Vec<Block>> = HashMap::new();

if old_size < new_cond.size() {
   if let Some(successor) = id_to_blocks.get(&current).map(|block| {
      block
         .iter()
         .map(|block| block.).id)
         .collect::<Vec<i32>>()
   }) {
      ...
   }
}
```

```
struct Block {
   id: usize,
}

let mut id_to_blocks: HashMap<usize, Vec<Block>> = HashMap::new();

fn blocks_to_ids(blocks: Vec<Block>) -> Vec<usize> {
   block
         .iter()
         .map(|block| block.borrow().id)
         .collect::<Vec<usize>>()
}

if old_size < new_cond.size() {
   if let Some(successor) = id_to_blocks.get(&current).map(blocks_to_ids) {
      ...
   }
}
```

使用 Trait 来表达代码的行为

```rust
pub struct BasicBlock<T: InstrTrait<U>, U: TempTrait> {
    pub id: i32,
    pub weight: f64,
    pub kill_size: i32,
    pub prev: Vec<Node<T, U>>,
    pub succ: Vec<Node<T, U>>,
    pub defs: HashSet<U>,
    pub uses: HashSet<U>,
    pub kills: HashSet<U>,
    pub live_in: HashSet<U>,
    pub live_out: HashSet<U>,
    pub phi_instrs: Vec<PhiInstr>,
    pub phi_defs: HashSet<LlvmTemp>,
    pub instrs: Vec<T>,
    pub jump_instr: Option<T>,
}
```

一个例子

```rust
#[derive(Debug, Copy, Clone)]
pub struct Point {
    x: i64,
    y: i64,
}

#[derive(Debug, Copy, Clone)]
pub struct Bounds {
    top_left: Point,
    bottom_right: Point,
}

/// Trait for objects that can be drawn graphically.
pub trait Draw {
    /// Return the bounding rectangle that encompasses the object.
    fn bounds(&self) -> Bounds;

    // ...
}
```

Trait Bound

在 C++ 模板的基础上，对模板类型 T 做出限制，规定 T 应当具有哪些方法，或者说，哪些特性（trait）

```
pub struct Bounds
pub trait Draw {
    fn bounds(&self) ->
Bounds;
}
```

三种写法：

```
pub fn on_screen<T>(draw: &T) ->
bool
where
    T: Draw,
{
    draw.bounds()
}
```

```
pub fn on_screen<T: Draw>(draw: &T) -> bool
{
    draw.bounds()
}
```

```
pub fn on_screen(draw: &impl Draw) -> bool {
    draw.bounds()
}
```

Trait Object

可以看成一种 fat pointer，指向的是某个 Trait 所包含的一组方法（函数）
体现的是动态多态的编程思想，其类型在运行时确定
形如 dyn Draw，由于编译期无法确定其大小，所以只能成为引用，或者通过 Box 将其分配在堆上

```
let square = Square {};
let draw: &dyn Draw = &square;
let box_draw: Box<dyn Draw> = Box::new(square.clone())
```

```
pub struct Bounds
pub trait Draw {
   fn bounds(&self) ->
Bounds;
}
```

Trait Bound v.s. Trait Object

1. Trait Bound 会生成更大的代码体积，因为它是静态多态，在编译期会为每一个类型生成一份代码
2. Trait Bound 比 Trait Object 会快一点点，因为 Trait Object 是动态多态，在运行时需要解两层引用（从 trait object 解引用获得 vtable, 再从 vtable 解引用获得实际实现
3. Trait Bound 会带来更长的编译耗时

但最大的区别其实是，Trait Bound 可以叠加，而 Trait Object 不能。你可以写出 where T: A + B, 而不能写出 dyn A+B

Option、Result 及相关的一系列成员函数

使用 Result 和 Option 进行异常处理

Option<T>: 表达类型为 T 的值可能存在，也可能不存在
——处理缺值的异常
Result<T, E>: 经常成为函数的返回类型。当函数表达的操作成功时，返回类型为 T 的值（Ok(T)）；当失败时，返回类型为 E 的值（(Err(E)）
——处理程序运行异常

永远使用它们来描述和处理异常！！！
可以将许多问题暴露在编写代码和编译阶段

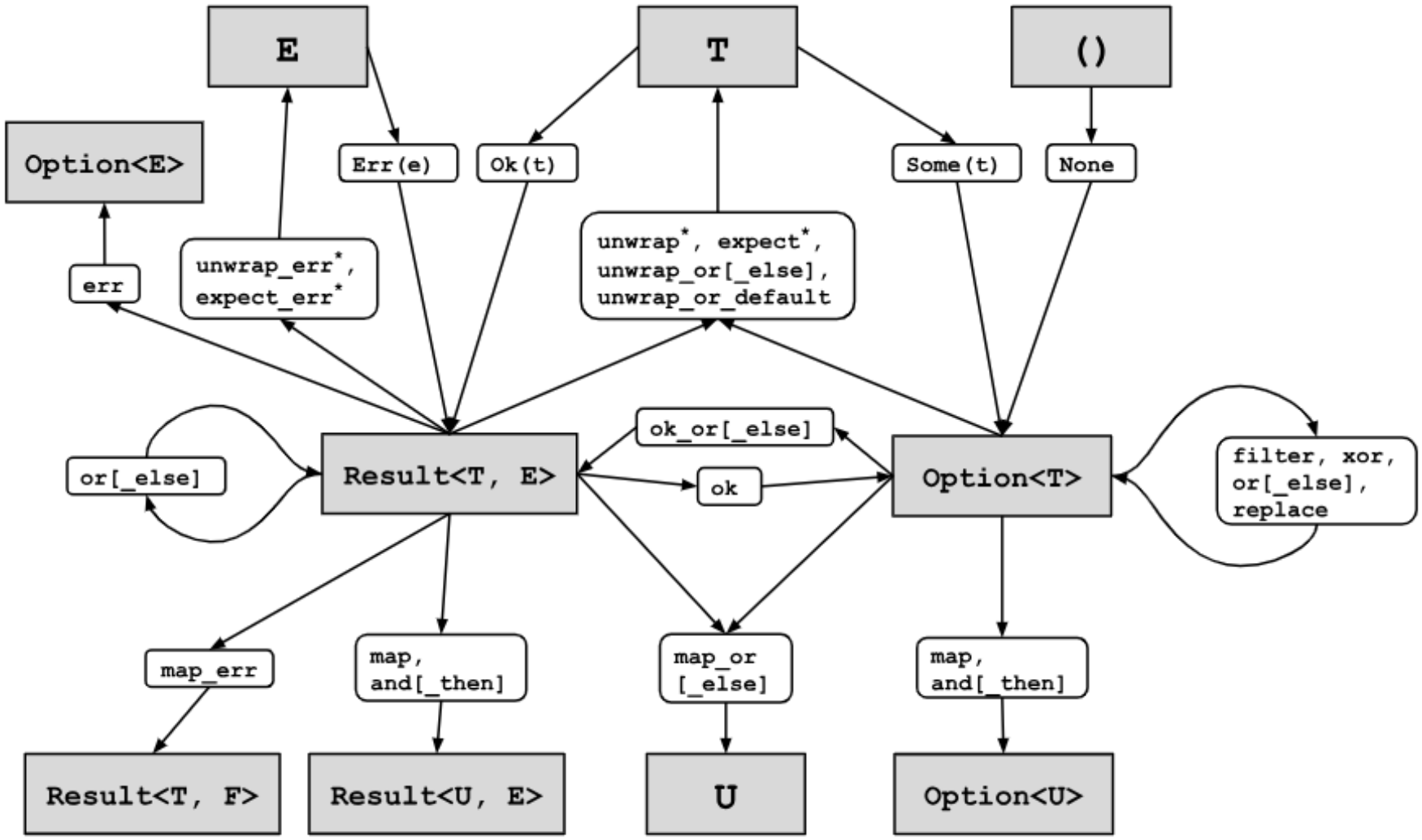
还记得 C++ 无缘无故的 core dumped 和 Segmentation Fault 吗：）

绕开模式匹配的一系列函数

Result 和 Option 本质上是 enum，直接使用会产生大量的模式匹配语句
Rust 标准库提供了一系列函数，帮助用户绕开模式匹配

map(), or(), ok_or(), unwrap_or(), and_then(), filter(), replace()...

我们 Rust 自己的 Transformer!



函数讲解

unwrap()

从 Result 或 Option 中获取值，如果是 Err 或 None 则 panic。

```
let a = Some(42)
a.unwrap() // 42
```

expect()

与 unwrap 类似，但可以提供自定义的 panic 消息。

```
let a = Some(42)
a.expect("core dumped") // 42
```

函数讲解

unwrap_or(T)

获取 Result 或 Option 中的值，如果是 Err(_) 或 None 则返回默认值（不会 panic！）。

```
let a = None;
let b = a.unwrap_or(30) // b == 30
```

unwrap_or_else(f)

获取 Result 或 Option 中的值，如果是 Err(_) 或 None 则调用提供的闭包来生成默认值（不会 panic！）。

```
let b = a.unwrap_or_else(|| 15+15) // b == 30
```

unwrap_or_default(T: Default)

获取 Result 或 Option 中的值，如果是 Err(_) 或 None 则返回类型的默认值，类型 T 需要实现 Default（不会 panic！）。

函数讲解

ok()

将 Result 转换为 Option，如果是 Ok(T) 则返回 Some(T)，如果是 Err(_) 则返回None。

```
let a = Ok(33)
let b = a.ok() // b == Some(33)
```

ok_or(E)

将 Option 转换为 Result，如果是 Some(T) 则返回 Ok(T)，如果是 None 则返回 Err(E)。

```
let a = None
let b = a.ok_or("Error!!") // b == Err("Error!!")
```

ok_or_else(f)

将 Option 转换为 Result，如果是 Some(T) 则返回 Ok(T)，如果是 None 则调用提供的闭包来生成 Err(E)。

函数讲解

filter(f)

对 Option 的值进行过滤，如果满足条件则返回 Some(T)，否则返回 None。

```
let some_number = Some(15);
let filtered_number = some_number.filter(|&x| x > 10);
println!("{:?}", filtered_number); // 输出: Some(15)
```

xor(Option<T>)

对两个 Option 进行异或运算。

```
let a = None;  let b = Some(10);
let result = a.xor(b); result == Some(10)
let a = Some(5); letb = Some(10);
let result = a.xor(b); result == None
```

函数讲解

replace(T)

替换Option的值并返回旧值。

```
let mut opt = Some(5);
let old_value = opt.replace(10);
// opt == Some(10); old_value == Some(5)

let mut opt = None;
letold_value = opt.replace(20);
// opt == ???; old_value == ???
```

函数讲解

or(T)

返回第一个非None的Option。

```
let a = Some(5);
let b = Some(10);
let result = a.or(b); // result == Some(5)
let a: Option<i32> = None;
let b = Some(10);
let result = a.or(b); // result == Some(10)
let a: Option<i32> = None;
let b: Option<i32> = None;
let result = a.or(b); result == None
```

函数讲解

or_else(f)

如果Option是None则调用提供的闭包来生成新的Option。

and_then(f)

对Option或Result中的值应用提供的闭包，并返回新的Option或Result。

```
let a = Some(5);
let result = a.and_then(|_| None::<i32>);
println!("{:?}", result); // 输出: None

let a: Option<i32> = None;
let result = a.and_then(|x| Some(x * 2));
println!("{:?}", result); // 输出: None
```

函数讲解

map(f)

对 Option 或 Result 中的值应用提供的闭包，并返回新的 Option 或 Result。

```
let a: Option<i32> = None;
let result = a.map(|x| x * 2);
println!("{:?}", result); // 输出: None
let a = Some(5);
let result = a.map(|x| x * 2);
println!("{:?}", result); // 输出: ???
```

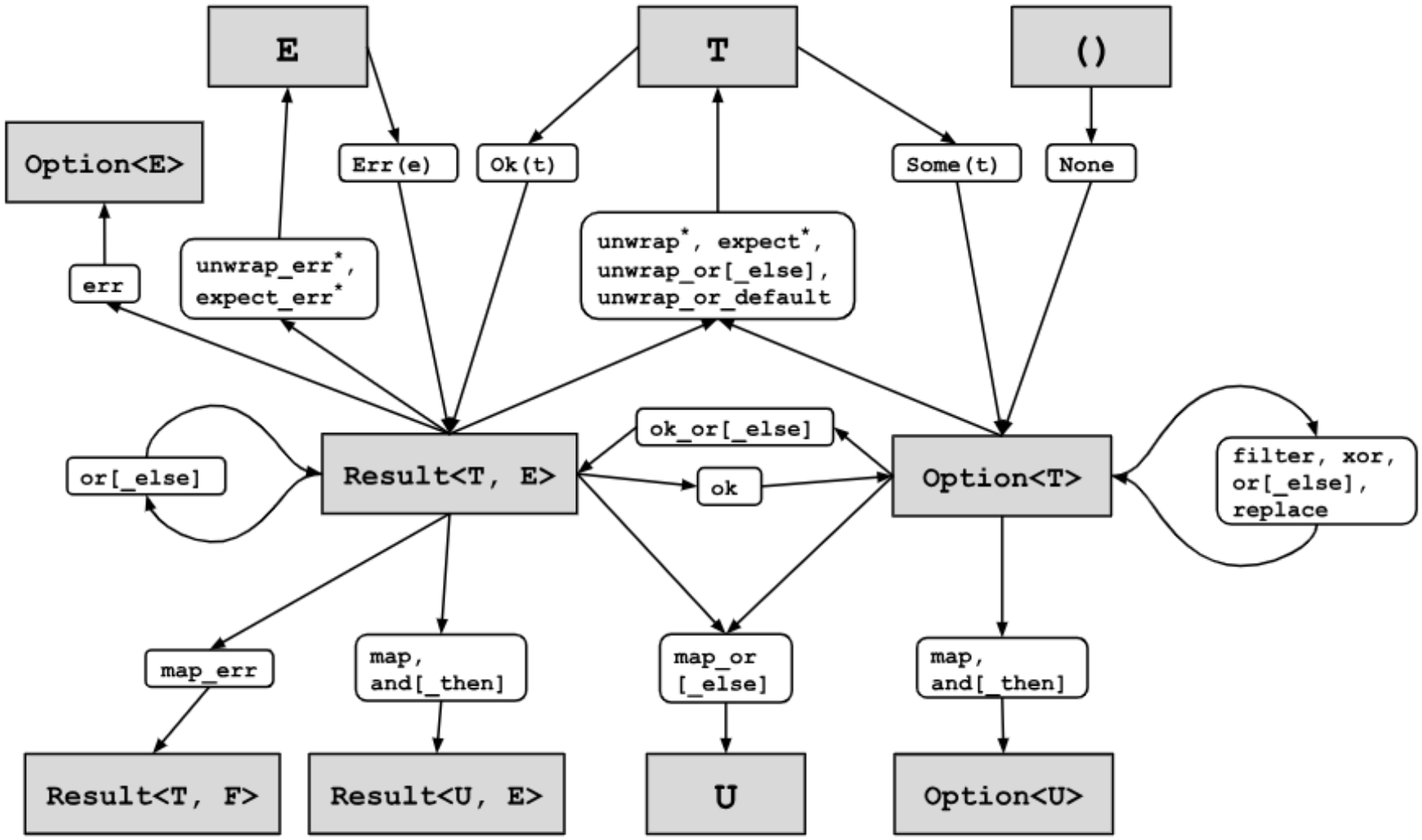
函数讲解

map_err(f)

对 Result 中的 Err 值应用提供的闭包，并返回新的 Result。如果是 Ok，则直接返回原值。

```
let a: Result<i32, &str> = Ok(5);
let result = a.map_err(|e| format!("Error: {}", e));
println!("{:?}", result); // 输出: Ok(5)
let a: Result<i32, &str> = Err("error");
let result = a.map_err(|e| format!("Error: {}", e));
println!("{:?}", result); // 输出: ???
```

我们 Rust 自己的 Transformer!



as_ref()

Option::as_ref 方法用于将 &Option<T> 转换为 Option<&T>。如果 Option 是 Some，则返回 Some 包含值的引用；如果是 None，则返回 None。
Result::as_ref 方法用于将 &Result<T, E> 转换为 Result<&T, &E>。如果 Result 是 Ok，则返回 Ok 包含值的引用；如果是 Err，则返回 Err 包含错误的引用。

Result 与错误处理

？运算符

对于 Result，如果？作用的表达式 Err，则会将这个 Err 提前返回。如果表达式是 Ok，则会解包 Ok 并将值返回给调用者。
当使用？运算符时，如果表达式返回 None，则会提前返回 None。如果返回 Some，则会解包 Some 并将值返回给调用者。

对 Option:

```
let b = a?;

let b = match a {
    Some(v) => v,
    None => return None,
}
```

对 Result:

```
let b = a?;

let b = match a {
    Ok(v) => v,
    Err(e)  => return Err(From::from(e)),
}
```

一个例子（来自我自己写的 OJ）

内容：当提交此任务的用户 id 大于总用户数量时，也就是用户不存在时，报错

目的：借助 ? 运算符使代码变简洁

现有问题：

1. 函数语义不单一（create_job 用于创建任务，但其中直接包含了检查用户是否存在的逻辑）
2. 似乎没有一个值为 Result 的表达式

```
#[post("/jobs")]
async fn create_job(job: web::Json<PostJob>) -> impl Responder {
    if job.user_id >= USERS.len() as u64 {
        return HttpResponse::NotFound().json(Error{
            code: 3,
            reason: String::from("ERR_NOT_FOUND"),
            message: String::from("User Not Found")
        })
    }
}
```

改进 1

函数拆分，语义更明确了

现有问题：
create_job 的返回值不是 Result
并且我们希望在出错的时候仍然返回 NotFound 响应

```
fn user_exists(user_id: u64) -> Result<(), Error>{
    if user_id >= USERS.len() as u64 {
        return Err(Error{
            code: 3,
            reason: String::from("ERR_NOT_FOUND"),
            message: String::from("User Not Found")
        })
    }
    Ok(())
}

#[post("/jobs")]
async fn create_job(job: web::Json<PostJob>) -> impl Responder  {
    user_exists(job.user_id)?;
}
```

改进 2

使用 actix_web::Result 和 actix_web::Error

现有问题：
报错：trait bound 'ResponseError' is not satisfied

```
fn user_exists(user_id: u64) -> Result<(), Error>{
    if user_id >= USERS.len() as u64 {
        return Err(Error{
            code: 3,
            reason: String::from("ERR_NOT_FOUND"),
            message: String::from("User Not Found")
        })
    }
    Ok(())
}

#[post("/jobs")]
async fn create_job(job: web::Json<PostJob>) -> Result<impl Responder, Error> {
    user_exists(job.user_id)?;
}
```

改进 2

为自定义 Error 类型实现 ResponseError，决定出错时 actix 框架应当返回怎样的响应

显然经过这样实现后，可以方便后续的错误处理

```
impl ResponseError for Error{
    fn error_response(&self) -> HttpResponse<actix_web::body::BoxBody> {
        HttpResponse::NotFound().json(self)
    }
}

fn user_exists(user_id: u64) -> Result<(), Error>{
    if user_id >= USERS.len() as u64 {
        return Err(Error{
            code: 3,
            reason: String::from("ERR_NOT_FOUND"),
            message: String::from("User Not Found")
        })
    }
    Ok(())
}

#[post("/jobs")]
async fn create_job(job: web::Json<PostJob>) -> Result<impl Responder, Error>  {
    user_exists(job.user_id)?;
}
```

迭代器及其相关函数

Rust 中的迭代器

迭代器（Iterator）是用于遍历集合（如数组、向量、哈希表等）元素的一种抽象方式。它提供了一种统一的方式来访问集合中的每一个元素，并允许对这些元素进行各种处理（如过滤、映射、累加等）。

Rust 中的迭代器：以两个 Trait 的形式存在

```
pub trait Iterator {
    type Item;
    fn next(&mut self) ->
Option<Self::Item>;
}
```

只需要实现 next 方法，告诉迭代器如何获得下一个元素

```
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator<Item = Self::Item>;
    fn into_iter(self) -> Self::IntoIter;
}
```

绑定一个已经实现了 Iterator trait 的类型，会夺走原数据的所有权

一个例子

```
struct Counter {
    count: u32,
}
impl Counter {
    fn new() -> Counter {
        Counter { count: 0 }
    }
}
```

```
// 为 Counter 实现 Iterator trait
impl Iterator for Counter {
    type Item = u32;
    fn next(&mut self) -> Option<Self::Item> {
        self.count += 1;
        if self.count <= 5 {
            Some(self.count)
        } else {
            None
        }
    }
}
```

```
// 为 Counter 实现
IntoIterator trait
impl IntoIterator for Counter {
    type Item = u32;
    type IntoIter = Counter;
    fn into_iter(self) ->
Self::IntoIter {
        self
    }
}
```

iter(), iter_mut()和into_iter()

这三个方法通常用于将集合类型转换为迭代器
如：Vec<T>::new().iter()
返回的 Iterator 的 Item = &T

.iter() 方法通常用于获取集合的不可变引用的迭代器。你可以遍历集合中的元素，但不能修改它们。
.iter_mut() 方法通常用于获取集合的可变引用的迭代器。你可以遍历集合中的元素，并能修改它们。
.into_iter() 方法通常用于获取集合的所有权的迭代器。你可以遍历集合中的元素，并且会转移其所有权。

链式调用

迭代器的链式调用（chaining）是一种强大且常用的编程模式。多个迭代器函数可以连续调用，每个函数都会返回一个新的迭代器，从而实现对数据的灵活且简洁的操作。迭代器函数是 Iterator trait 中被自动实现的方法，返回值的类型也是 impl Iterator。

```
let values: Vec<u64> = vec![1, 1, 2, 3, 5 /* ... */];
let even_sum_squares: u64 = values.iter().filter(|x| *x % 2 == 0).take(5).map(|x| x * x).sum();
```

迭代器函数有：map, chain, filter, flatten, for_each,

for_each()

for_each 方法对迭代器中的每个元素应用一个闭包。这通常用于具有副作用的操作，比如打印每个元素。

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    numbers.iter().for_each(|&x| println!("{}", x));
    // 输出:
    // 1
    // 2
    // 3
    // 4
    // 5
}
```

它实际上和 for ... in ....iter() 是等效的

map()

map 方法对迭代器中的每个元素应用一个闭包，并返回一个新的迭代器。这个新的迭代器包含闭包的返回值。

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let doubled: Vec<i32> = numbers.iter().map(|&x| x * 2).collect();
    println!("{:?}", doubled); // 输出: [2, 4, 6, 8, 10]
}
```

chain()

chain 方法将两个迭代器连接在一起，返回一个新的迭代器。这个新迭代器会先遍历第一个迭代器，然后遍历第二个迭代器。

```rust
fn main() {
    let numbers1 = vec![1, 2, 3];
    let numbers2 = vec![4, 5, 6];
    let chained: Vec<i32> = numbers1.iter().chain(numbers2.iter()).cloned().collect()
    println!("{:?}", chained); // 输出: [1, 2, 3, 4, 5, 6]
}
```

zip()

zip 方法将两个迭代器配对在一起，返回一个新的迭代器。这个新迭代器的元素是元组，包含两个迭代器的对应元素。

```rust
let numbers = vec![1, 2, 3];
let letters = vec!['a', 'b', 'c'];
let zipped: Vec<(i32, char)> = numbers.iter().zip(letters.iter()).cloned().collect();
println!("{:?}", zipped); // 输出: [(1, 'a'), (2, 'b'), (3, 'c')]
```

enumerate()

enumerate 方法为迭代器中的每个元素加上一个索引，返回一个新的迭代器。这个新迭代器的元素是元组，包含元素的索引和元素的值。

```rust
fn main() {
    let numbers = vec![1, 2, 3];
    for (index, value) in numbers.iter().enumerate() {
        println!("Index: {}, Value: {}", index, value);
    }
    // 输出:
    // Index: 0, Value: 1
    // Index: 1, Value: 2
    // Index: 2, Value: 3
}
```

filter()

filter 方法对迭代器中的每个元素应用一个闭包，只保留满足条件的元素，返回一个新的迭代器。

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];
    let even_numbers: Vec<i32> = numbers.iter().filter(|&&x| x % 2 == 0).collect();
    println!("{:?}", even_numbers); // 输出: [2, 4]
}
```

retain()

retain 方法用于直接修改原始集合，保留满足条件的元素。它不会返回一个新的迭代器，而是直接在原集合上进行操作。

```rust
fn main() {
    let mut numbers = vec![1, 2, 3, 4, 5];
    numbers.retain(|&x| x % 2 == 0);
    println!("{:?}", numbers); // 输出: [2, 4]
}
```

collect()

collect 函数用于将迭代器转换为集合（例如 Vec, HashSet, HashMap 等）。collect 函数通过实现 FromIterator trait 来实现这一功能。

```rust
fn main() {
    let numbers = vec![1, 2, 3, 4, 5];

    // 使用迭代器过滤偶数并收集到一个新的向量中
    let even_numbers: Vec<i32> = numbers.iter().cloned().filter(|&x| x % 2 == 0).collect();

    println!("{:?}", even_numbers); // 输出: [2, 4]
}
```

其余方法

- map
- filter
- filter_map
- flat_map
- skip
- skip_while
- take
- take_while
- enumerate
- peekable
- fuse
- inspect
- cloned
- copied
- collect
- partition
- unzip
- fold
- for_each
- reduce
- scan
- by_ref
- count
- sum
- chain
- cycle
- zip
- rev
- product
- max
- min
- max_by
- try_fold
- try_for_each
- advance_by
- filter_count
- min_by
- max_by_key
- min_by_key
- find
- find_map
- position
- rposition
- any
- all
- nth
- nth_back
- last

在未来 Rust 可以帮到你的课程

在未来 Rust 可以帮到你的课程

- 操作系统: 使用 C++ 编写 ucore 和使用 Rust 编写 rcore 二选一
- 编译原理大实验: 使用 Rust 写编译器
- 软件工程: 使用 Rust 编写后端（Rust 写前端应该还不太好用吧，，，）
- 数据库系统概论：Rust 和 C++ 二选一，Rust 可以解决你的所有内存问题
- ~~密码学~~
- 分布式系统导论的课程实验（研究生）：Go 和 Rust 二选一

谢谢大家！

陈英豪

2024 年 7 月 17 日