

# Rust 高级特性

Rust 开发实训

---

王杰/穿山甲

2023/09/04

# CONTENT

## 目录

01. Trait
02. Lifetime
03. Macro

# CONTENTS

## 目录

-  04. Smart Pointers
-  05. Fearless Concurrency

# 01

## Trait

A trait defines functionality a particular type has and can share with other types

# Define a trait



## Definition

Trait是一种实现接口和代码复用的方式。Trait表示一组方法（有时也包括默认实现），作为一种行为或能力的抽象。类似于其他编程语言中的接口，例如Java中的接口（Interface）、C++中的抽象类（Abstract Classes）、Python中的抽象基类（Abstract Base Classes）、Go 中的接口类型（Interface Type）等



## Example

```
trait Sound {  
    fn make_sound(&self);  
}
```

# Implementing a Trait on a Type

我们为Dog、Cat和Bird结构体实现了Sound，这样就可以为不同的动物类型调用make\_sound方法，从而模拟不同动物的叫声

```
fn main() {
    let dog = Dog { name: String::from("Buddy") };
    let cat = Cat { name: String::from("Kitty") };
    let bird = Bird { name: String::from("Tweety") };

    dog.make_sound();
    cat.make_sound();
    bird.make_sound();
}
```

```
struct Dog {
    name: String,
}

struct Cat {
    name: String,
}

struct Bird {
    name: String,
}

// 为各种动物实现 Sound trait
impl Sound for Dog {
    fn make_sound(&self) {
        println!("{} says: Woof!", self.name);
    }
}

impl Sound for Cat {
    fn make_sound(&self) {
        println!("{} says: Meow!", self.name);
    }
}

impl Sound for Bird {
    fn make_sound(&self) {
        println!("{} says: Chirp!", self.name);
    }
}
```

# Default Implementations

我们也可以为Trait中的方法加上默认实现

```
trait Sound {  
    fn make_sound(&self) {  
        println!("The animal makes a sound")  
    }  
  
    struct Fish {  
        name: String,  
    }  
  
    impl Sound for Fish {} // 不重写，使用默认实现  
  
    fn main() {  
        let fish = Fish { name: String::from("fish") };  
        // 调用默认实现的make_sound方法  
        fish.make_sound();  
    }  
}
```

# Traits as Parameters

可以使用Impl Trait语法作为函数参数和返回值的类型，这种语法提供了鸭子类型（Duck typing），它允许在不显式声明具体类型的情况下，根据实现的trait来定义输入或输出的约束

```
fn emit_sound(animal: impl Sound) {  
    animal.make_sound();  
}  
  
fn create_dog(name: &str) -> impl Sound {  
    Dog {  
        name: name.to_string(),  
    }  
}  
  
fn main() {  
    let dog_name = "Buddy";  
    let dog = create_dog(dog_name);  
    emit_sound(dog);  
  
    let cat = Cat { name: String::from("Kitty") };  
    emit_sound(cat);  
}
```

# Traits Bound Syntax

Impl Trait实际上是Trait Bound的一种语法糖，等价于右图写法

```
// 使用泛型和trait bounds作为参数
fn emit_sound<T: Sound>(animal: T) {
    animal.make_sound();
}

fn main() {
    let dog_name = "Buddy";
    let dog = create_dog(dog_name);
    emit_sound(dog);
}
```

# Specifying Multiple Trait Bounds with the + Syntax

我们可以通过符号 + 来指定多个Trait Bound

```
fn emit_sound<T: Sound + Display>(animal: T) {  
    println!("display animal: {:?}", animal);  
    animal.make_sound();  
}
```

# Associated Types

关联类型是将特定类型与特定trait关联的方法。通过为trait使用type关键字定义关联类型，我们可以使该trait的实现更加灵活，同时减少不必要的类型参数

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

# Default Generic Type

可以为泛型类型参数定义默认类型，这样，在该类型未显式指定时，编译器会自动使用默认类型，使得实现接口更加简洁和灵活

```
trait Add<Rhs = Self> {  
    type Output;  
    fn add(self, rhs: Rhs) -> Self::Output;  
}
```

# Fully Qualified Syntax

在某些情况下，当一个类型实现了多个trait，其中一些trait具有相同的方法名时，直接调用该方法可能会引发“歧义”。在这种情况下，可以使用完全限定语法显式指定调用哪个trait的方法

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct S;

impl Foo for S {
    fn f(&self) {
        println!("S: Foo");
    }
}

impl Bar for S {
    fn f(&self) {
        println!("S: Bar");
    }
}

fn main() {
    let s = S;
    Foo::f(&s);
    Bar::f(&s);
}
```

# Super Traits

有时候，实现一个trait可能需要另一个trait的功能，此时，我们可以将该先决条件trait定义为超trait。这样可以确保在实现当前trait之前，类型必须实现超trait

```
trait Foo {  
    fn foo(&self);  
}  
  
trait Bar: Foo {  
    fn bar(&self);  
}
```

# Newtype Pattern

有时，向外部类型添加trait实现可能引起孤儿规则问题，为了避免这种情况，可以使用新类型模式：为现有类型起一个新的别名，并为其实现所需的trait。通过在结构体中封装原类型来实现这一点

```
struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}
```

# Test

为Vec<T>实现一个Summable trait，其中包含一个名为sum的方法，  
该方法返回T的总和。要求支持整数和浮点数类型

# Test

```
trait Summable {
    type Output;
    fn sum(&self) -> Self::Output;
}

impl Summable for Vec<i32> {
    type Output = i32;

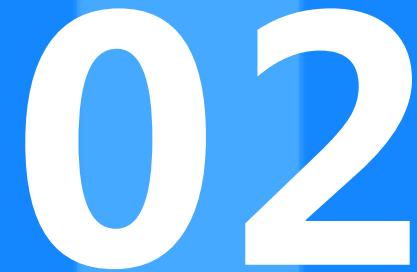
    fn sum(&self) -> i32 {
        self.iter().sum()
    }
}

impl Summable for Vec<f64> {
    type Output = f64;

    fn sum(&self) -> f64 {
        self.iter().sum()
    }
}

fn main() {
    let integers = vec![1, 2, 3, 4];
    let floats = vec![1.1, 2.2, 3.3, 4.4];

    println!("Sum of integers: {}", integers.sum());
    println!("Sum of floats: {}", floats.sum());
}
```



02

## Lifetime

Lifetimes ensure that references are valid as long as we need them to be

## Definition

Lifetime（生命周期）是一个编译时的概念，用于确保引用类型的有效性。生命周期允许Rust在编译时进行借用检查，以确保引用类型访问的资源在其整个生命周期内仍然有效。可以防止出现悬垂指针等问题，即在一个资源被释放后仍然有指针或引用指向它

# Example

下图示例中，`r`的生命周期为 '`a`'，`x`的生命周期为 '`b`'，且 '`a > b`'，`r`引用了`x`，在`x`被释放之后，`r`仍然引用`x`，则会出现ub(未定义行为)。Rust通过Borrow Checker在编译期检查这些情况(引用是否合法)，以下代码无法通过编译

```
fn main() {
    let r; // -----
    // |
    {
        let x = 5; // -+-- 'b
        r = &x; // |
    } // ++
    // |
    // |
    println!("r: {}", r); // -----
}
```

## Example

改成以下写法后，引用正常，可以通过编译。也即，被引用者的生命周期务必不小于引用者的生命周期，此处为 '`b` > '`a`'

```
fn main() {
    let x = 5;           // -----+-- 'b
    let r = &x;          // ---+-- 'a |
    println!("r: {}", r); //   | |
}                                // -----+|
```

## Example

下图示例中，我们定义了一个longest函数，它用来返回s1、s2之间较长者，但无法通过编译。因为编译器无法确定返回值是s1还是s2，也就无法确定返回值的生命周期

```
fn longest<'a>(s1: &str, s2: &str) -> &str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

## Example

我们需要显示的标注出来出入参的生命周期，如下图所示，我们使用 '' 来表示一个生命周期，也即告诉编译器出入参的生命周期是一样的，这样便可通过编译

```
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {  
    if s1.len() > s2.len() {  
        s1  
    } else {  
        s2  
    }  
}
```

## Example

以下代码是否存在问题？

```
fn main() {
    let long_string = String::from("This is a long string.");
    let result;
    {
        let short_string = String::from("Short");
        result = longest(&long_string, &short_string);
    }
    println!("The longest string is: {}", result);
}
```

# Test

以下代码是否存在问题？

```
struct Interface<'a> {
    manager: &'a mut Manager<'a>
}

impl<'a> Interface<'a> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'a> {
    text: &'a str
}

struct List<'a> {
    manager: Manager<'a>|,
}

impl<'a> List<'a> {
    pub fn get_interface(&'a mut self) -> Interface {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // this fails because immutable/mutable borrow
    // but Interface should be already dropped here and the borrow released
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}
```

# Test

指定适当的生命周期后，可通过编译

```
struct Interface<'text, 'manager> {
    manager: &'manager mut Manager<'text>
}

impl<'text, 'manager> Interface<'text, 'manager> {
    pub fn noop(self) {
        println!("interface consumed");
    }
}

struct Manager<'text> {
    text: &'text str
}

struct List<'text> {
    manager: Manager<'text>;
}

impl<'text> List<'text> {
    pub fn get_interface<'manager>(&'manager mut self) -> Interface<'text, 'manager>
    where 'text: 'manager {
        Interface {
            manager: &mut self.manager
        }
    }
}

fn main() {
    let mut list = List {
        manager: Manager {
            text: "hello"
        }
    };

    list.get_interface().noop();

    println!("Interface should be dropped here and the borrow released");

    // now, it's ok
    use_list(&list);
}

fn use_list(list: &List) {
    println!("{}", list.manager.text);
}
```

# 03

## Macro

A way of writing code that writes other code

## Definition

宏允许在编译时生成代码，是一种元编程技术，可以减少我们所需要编写的代码。

Rust提供了两种主要类型的宏：声明式宏、过程宏

## Example

我们可能会通过如下方式来创建并初始化一个Vector，在初始化元素可确定的情况下，是否存在更为简便的方法？

```
fn main() {
    let mut v = Vector::new();
    v.push(1);
    v.push(2);
    v.push(3);
    // ...
}
```

## Example

或许我们可以定义一个函数？但遗憾的是，Rust不支持可变长参数，也即其参数数量应当是确定的。换个思路，这一段重复的代码，我们可以使之在编译期生成，我们只要提供类似函数的入参，剩下的交给编译器即可，也即，宏

## Example

我们可以通过macro\_rule!来定义一个声明式宏，类似于match(模式匹配)

```
macro_rules! macro_name {
    (pattern1) => {
        },
    (pattern2) => {
        },
    // ...
}
```

## Example

现在我们来尝试定义一个简单的宏，用来输出一条语句。如下，我们定义了一个名为hello的宏，用来输出一条语句。仅存在一条分支，匹配参数为空，我们通过hello!()的方式来调用它

```
macro_rules! hello {
    () => {
        println!("Hello, World!")
    }
}

fn main() {
    hello!();
}
```

# Example

回到之前的问题，我们现在来定义一个宏，用来帮助我们创建并初始化一个Vector集合

```
macro_rules! my_vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

# Example

我们来调用它，传入我们希望的初始化的参数即可

```
fn main() {  
    let v = my_vec![1, 2, 3];  
}
```

# Example

事实上，我们调用宏的这一行代码会在编译器被替换成以下的代码段

```
fn main() {  
    let v = my_vec![1, 2, 3];  
  
    let v = {  
        let mut temp_vec = Vec::new();  
        temp_vec.push(1);  
        temp_vec.push(2);  
        temp_vec.push(3);  
        temp_vec  
    };  
}
```

# Procedural Macros

过程宏是一种更高级的宏类型，它允许我们编写可以操作 Rust 语法树（AST）的代码。

过程宏分为三类：派生宏（Derive Macros）、属性宏（Attribute Macros）和函数宏

## Derive Macros

允许自动为类型实现某些 trait，例如，通过 #[derive(Debug)] 自动生成类型的 Debug trait 实现

```
#[derive(Debug)]
struct Example {

}
```

# Attribute Macros

定义了可以应用于结构体、函数或模块等项的自定义属性。例如，#[serde(rename = "x")] 用于改变 Serde 序列化库中字段的名称

```
#[serde(serde::Serialize, serde::Deserialize)]
struct Example {
    #[serde(rename = "x")]
    field1: String
}
```

# Function-like Macros

这种宏类似于 macro\_rules! 定义的宏，但以函数的形式编写。它们可以直接操作 Rust 语法树（AST），在编译时生成更复杂的代码

```
fn main() {  
    let sql = sql!(SELECT * FROM posts WHERE id=1);  
}
```

# Writing Procedural Macros

我们可以通过以下方式来编写一个派生宏，其他宏也是类似的，使用不同的属性即可，如proc\_macro\_attribute、proc\_macro

```
use proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}

fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        };
    gen.into()
}
```

# Test

实现一个名为hash\_map!的宏，它接受偶数个参数，并生成一个  
std::collections::HashMap，其中第一个参数是键，第二个参数是值，以此类推

# Test

```
#[macro_use]
extern crate std;

use std::collections::HashMap;

macro_rules! hash_map {
    ($($key:expr => $val:expr),*) => {
        {
            let mut map = HashMap::new();
            $($(
                map.insert($key, $val);
            )*)
            map
        }
    };
}

fn main() {
    let map = hash_map! {
        "one" => 1,
        "two" => 2,
        "three" => 3
    };

    println!("{:?}", map);
}
```

# 04

## Smart Pointers

Data structures that act like a pointer but also have additional metadata and capabilities

## Definition

智能指针（Smart Pointers）是一种更高级的指针类型，它不仅提供了对内存的访问，还具有额外的元数据和功能。智能指针通常实现了特定的 Rust traits，例如 Deref 和 Drop，以提供自动解引用和资源清理等特性。这些智能指针有助于实现更安全、更方便的内存管理和所有权模型。

# Why

## 1. 自动内存管理

智能指针可以自动管理内存分配和释放，减轻了手动管理内存的负担。例如，`Box<T>`会在离开作用域时自动释放分配的堆内存

## 2. 引用计数

某些智能指针（如 `Rc<T>` 和 `Arc<T>`）提供了引用计数功能，允许在多个地方共享数据，而无需担心内存泄漏。引用计数指针在最后一个引用离开作用域时自动释放内存

## 3. 内部可变性

智能指针（如 `RefCell<T>` 和 `Mutex<T>`）提供了内部可变性，允许在具有不可变引用的情况下修改数据。这有助于实现更灵活的数据共享策略，特别是在多线程环境中

## 4. 线程同步

Rust 提供了一些智能指针（如 `Mutex<T>` 和 `RwLock<T>`），用于在多线程环境中同步数据访问。这些智能指针可以确保在线程间共享数据时遵循正确的同步规则

## 5. 自定义析构逻辑

通过为智能指针实现 `Drop trait`，可以在离开作用域时自动执行一些清理和资源回收操作。这有助于确保即使在异常情况下，资源也能得到正确的释放

## Box<T>

Box<T> 是一个分配在堆上的智能指针。它在编译时大小固定，允许将大型数据结构或实现递归类型（如链表和树）存储在堆上。当 Box<T> 离开作用域时，它会自动释放分配的内存

```
fn main() {
    let b = Box::new(42);
    println!("Box value: {}", b);
}
```

Rc<T>（引用计数指针）是一个多所有权的智能指针，用于在多个地方共享只读数据。当最后一个引用离开作用域时，Rc<T> 会自动释放分配的内存。请注意，Rc<T> 不是线程安全的

```
use std::rc::Rc;

fn main() {
    let rc1 = Rc::new(42);
    let rc2 = Rc::clone(&rc1);
    println!("rc1: {}, rc2: {}", rc1, rc2);
}
```

## Arc<T>

Arc<T>（原子引用计数指针）与 Rc<T>类似，但它是线程安全的。Arc<T>在多线程环境中共享只读数据时非常有用

```
use std::sync::Arc;
use std::thread;

fn main() {
    let arc1 = Arc::new(42);
    let arc2 = Arc::clone(&arc1);

    let handle = thread::spawn(move || {
        println!("arc2: {}", arc2);
    });

    println!("arc1: {}", arc1);
    handle.join().unwrap();
}
```

# RefCell<T> & Mutex<T>

RefCell<T> 和 Mutex<T> 是两种提供内部可变性的智能指针。它们允许在运行时借用可变引用，而不是在编译时。RefCell<T> 只能用于非线程安全场景，Mutex<T> 用于线程安全场景。这些智能指针帮助管理内存分配、共享和同步，同时遵循 Rust 的所有权和借用规则，以确保内存安全

```
use std::cell::RefCell;

fn main() {
    let data = RefCell::new(42);

    // 获取不可变引用
    let x = data.borrow();
    println!("x: {}", x);

    // 获取可变引用
    {
        let mut y = data.borrow_mut();
        *y += 1;
    }

    // 再次获取不可变引用
    let z = data.borrow();
    println!("z: {}", z);
}
```

# Defining Our Own Smart Pointer

我们使用元组struct定义一个智能指针，如下

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

# Defining Our Own Smart Pointer

来尝试使用一下

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

```
$ cargo run
Compiling deref-example v0.1.0 (file:///projects/deref-example)
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
--> src/main.rs:14:19
|
14 |     assert_eq!(5, *y);
    |             ^
For more information about this error, try `rustc --explain E0614`.
error: could not compile `deref-example` due to previous error
```

# Defining Our Own Smart Pointer

我们需要为它实现Deref Trait，这样在我们使用\*y的时候，Rust编译器会自动调用deref()方法，也即\*(y.deref())得到T类型

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

# Test

实现一个简易的引用计数智能指针 MyRc，类似于std::rc::Rc

# Test

实现一个简易的栈（LIFO）数据结构，支持push和pop操作，使用RefCell来实现内部可变性

# Test

```
use std::cell::RefCell;

#[derive(Debug)]
struct SimpleStack<T> {
    stack: RefCell<Vec<T>>,
}

impl<T> SimpleStack<T> {
    fn new() -> SimpleStack<T> {
        SimpleStack {
            stack: RefCell::new(Vec::new()),
        }
    }

    fn push(&self, value: T) {
        self.stack.borrow_mut().push(value);
    }

    fn pop(&self) -> Option<T> {
        self.stack.borrow_mut().pop()
    }
}

fn main() {
    let stack = SimpleStack::new();
    stack.push(1);
    stack.push(2);
    stack.push(3);

    println!("Popped value: {:?}", stack.pop());
    println!("Popped value: {:?}", stack.pop());

    stack.push(4);

    println!("Popped value: {:?}", stack.pop());
    println!("Popped value: {:?}", stack.pop());
    println!("Popped value: {:?}", stack.pop());
}
```



05

## Fearless Concurrency

Easily write concurrent code without worrying  
about common concurrency issues

## Definition

无畏并发（Fearless Concurrency）是指 Rust 语言提供的一组功能和设计原则，使我们能够轻松编写并发代码，而无需担心常见的并发问题

# Ownership System

Rust 的所有权系统确保在任何时刻，要么只有一个可变引用，要么只有多个不可变引用。这样一来，在编译时就可以消除数据竞争的可能性

# Borrow Checker

Rust 的借用检查器确保引用满足所有权规则。通过在编译时强制执行这些规则，Rust 可以确保并发代码在运行时不会出现数据竞争、悬挂指针等问题

# Thread Primitives

Rust 标准库提供了一组线程原语（如 std::thread、std::sync::Mutex、std::sync::Arc 等），用于创建线程、同步数据和管理线程间的通信。这些原语基于 Rust 的所有权和借用规则，因此可以安全地用于编写并发代码

# Send、Sync Trait

Rust 中的 Send 和 Sync trait 用于表示跨线程共享的类型。如果一个类型实现了 Send，那么它可以安全地在线程间传递所有权。如果一个类型实现了 Sync，那么它可以安全地在线程间共享只读引用。这些 trait 自动为满足条件的类型实现，从而确保跨线程共享数据的安全性

# Example

```
use std::sync::Arc;
use std::thread;

fn main() {
    let data = Arc::new(vec![1, 2, 3, 4, 5]);
    let mut handles = vec![];

    for _ in 0..3 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            println!("Data in thread: {:?}", data);
        });
        handles.push(handle);
    }

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

## Example

使用 `std::sync::mpsc` 和线程实现一个简单的消息传递系统，模拟多个生产者和消费者之间的消息传递

# Example

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

fn main() {
    const PRODUCERS: usize = 3;
    const CONSUMERS: usize = 3;
    const MESSAGES_PER_PRODUCER: usize = 5;

    let (tx, rx) = mpsc::channel();
    let rx = Arc::new(Mutex::new(rx));

    let mut producer_handles = vec![];

    for i in 0..PRODUCERS {
        let tx = tx.clone();
        let handle = thread::spawn(move || {
            for j in 0..MESSAGES_PER_PRODUCER {
                let msg = format!("Producer {}: message {}", i, j);
                tx.send(msg).unwrap();
            }
        });
        producer_handles.push(handle);
    }
}
```

```
let mut consumer_handles = vec![];

for i in 0..CONSUMERS {
    let rx = Arc::clone(&rx);
    let handle = thread::spawn(move || {
        loop {
            let msg = match rx.lock().unwrap().recv() {
                Ok(m) => m,
                Err(_) => break,
            };
            println!("Consumer {}: {}", i, msg);
        }
    });
    consumer_handles.push(handle);
}

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

drop(tx);

for handle in consumer_handles {
    handle.join().unwrap();
}
}
```

THANKS

