

1.11 Rust 语法面面观：函数与闭包（上）



扫码试看/订阅

《张汉东的Rust实战课》视频课程

Rust 中的函数与闭包

内容包括：

1. 常规函数

2. 函数指针

3. 闭包

常规函数 / 特点

1. 函数都拥有显式的类型签名
2. 函数可以分为三种类型：自由函数、关联函数 和 方法
3. 函数自身也是一种类型

常规函数 / 自由函数

```
// 自由函数
fn sum(a: i32, b: i32) -> i32 {
    a+b
}

fn main(){
    assert_eq!(3, sum(1, 2))
    // sum(1, "2"); // error: 不满足函数签名的类型约定
}
```

常规函数 / 关联函数与方法

```
struct A(i32, i32);
impl A{
    // 关联函数
    fn sum(a: i32, b: i32) -> i32 {
        a+b
    }
    // 方法
    fn math(&self) -> i32 {
        Self::sum(self.0, self.1)
    }
}
fn main(){
    let a = A(1, 2);
    assert_eq!(3, A::sum(1, 2));
    assert_eq!(3, a.math());
}
```

常规函数 / 函数项类型

```
struct A(i32, i32);
impl A{
    fn sum(a: i32, b: i32) -> i32 {
        a+b
    }
    fn math(&self) -> i32 {
        Self::sum(self.0, self.1)
    }
}
fn main(){
    let a = A(1, 2);
    let add = A::sum;          // A::sum 是一个 Fn item 类型
    let add_math = A::math;    // A::math 也是一个 Fn item 类型
    assert_eq!(add(1, 2), A::sum(1, 2));
    assert_eq!(add_math(&a), a.math());
}
```


常规函数 / 零大小类型的类型构造器

```
enum Color {  
    R(i16),  
    G(i16),  
    B(i16),  
}  
  
// 等价于  
// fn Color::R(_1: i16) -> Color { /* ... */}  
// fn Color::G(_1: i16) -> Color { /* ... */}  
// fn Color::B(_1: i16) -> Color { /* ... */}  
// 零大小类型  
fn main(){  
    println!("{:?}", std::mem::size_of_val(&Color::R) ); // 0  
}
```

函数项默认实现了 Copy/Clone/Sync/Send/Fn/FnMut/FnOnce

函数指针

/ 函数项隐式转换为函数指针

```
type RGB = (i16, i16, i16);
```

```
fn color(c: &str) -> RGB {  
    (1,1,1)  
}
```

函数指针类型

```
fn show(c: fn(&str)-> RGB) {  
    println!("{:?}", c("black"));  
}
```

```
fn main(){
```

```
    let rgb = color;
```

函数项类型

```
    show(rgb); // (1,1,1)
```

```
}
```

函数项类型

函数指针 / 函数项隐式转换为函数指针

```
let rgb = color;    // 函数项类型 ( Fn item Type )  
let c: fn(&str)->RGB = rgb; // 隐式转换为了函数指针类型 (fn pointer Type)
```

```
println!("{:?}", std::mem::size_of_val(&rgb)); // 0  
println!("{:?}", std::mem::size_of_val(&c));    // 8
```

函数指针 / 结论

1. 函数项类型可以通过显式指定函数类型转换为一个函数指针类型
2. 在写代码的时候，尽可能地去使用 函数项类型，不到万不得已不要使用函数指针类型，这样有助于享受零大小类型的优化

闭包 / 函数无法捕获环境变量

```
fn counter(i: i32) -> fn(i32) -> i32 {  
    fn inc(n: i32) -> i32 {  
        n + i // error[E0434]: can't capture dynamic environment in a fn item  
    }  
    inc  
}  
  
fn main() {  
    let f = counter(2);  
    assert_eq!(3, f(1));  
}
```

闭包 / 闭包可以捕获环境变量

```
fn counter(i: i32) -> impl FnMut(i32) -> i32 {  
    move |n| n + i  
}  
  
fn main() {  
    let mut f = counter(2);  
    assert_eq!(3, f(1));  
}
```

闭包 / 闭包与函数指针互通

```
type RGB = (i16, i16, i16);
fn color(c: &str) -> RGB {
    (1,1,1)
}
fn show(c: fn(&str)-> RGB) {
    println!("{}", c("black"));
}
fn main(){
    let rgb = color;
    show(rgb); // (1,1,1)
    // 定义了实现 `Fn(&str) -> RGB` trait 的闭包类型
    let c = |s: &str| { (1,2,3) };
    show(c); // (1, 2, 3)
}
```

小结

1.12 Rust 语法面面观：函数与闭包（中）

深入学习Rust 中的闭包

内容包括：

1. Rust 闭包的实现原理

2. 闭包的具体分类

Rust 闭包的实现原理

/ 按使用场景分类

1.未捕捉环境变量

2.捕捉但修改环境变量

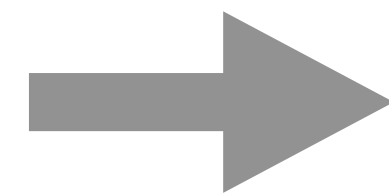
3.捕捉但未修改环境变量

```
fn main() {  
    // 未捕捉环境变量  
    let c1 = || println!("hello");  
    c1();  
  
    // 可修改环境变量  
    let mut arr = [1, 2, 3];  
    let mut c2 = |i| {  
        arr[0] = i;  
        println!("{:?}", arr);  
    };  
    c2(0);  
  
    // 未修改环境变量  
    let answer = 42;  
    let c3 = || {  
        println!(  
            "The answer to the Ultimate Question of Life, The Universe, and Everything is {}",  
            answer  
        );  
    };  
    c3();  
}
```

Rust 闭包的实现原理

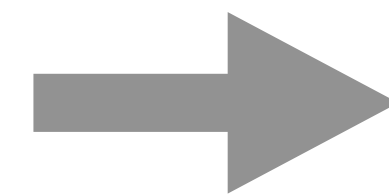
/ 与所有权语义三件套匹配

1.未捕捉环境变量



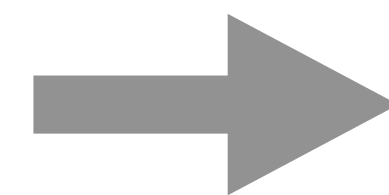
所有权 (Ownership)

2.捕捉但修改环境变量



可变借用 (&mut T)

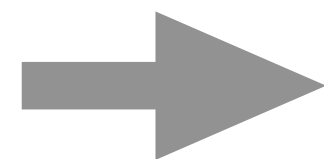
3.捕捉但未修改环境变量



不可变借用 (&T)

Rust 闭包的实现原理 / 第一类场景

```
fn main() {  
    // 未捕捉环境变量  
    let c1 = || println!("hello");  
    c1();  
}
```



```
#![feature(unboxed_closures, fn_traits)]  
struct Closure<T> {  
    env_var: T,  
}  
  
/**  
    ### 标准库 FnOnce trait 的定义  
    pub trait FnOnce<Args> {  
        type Output;  
        extern "rust-call" fn call_once(self, args: Args) -> Self::Output;  
    }  
*/  
impl<T> FnOnce<()> for Closure<T> {  
    type Output = ();  
    extern "rust-call" fn call_once(self, args: ()) -> () {  
        println!("hello");  
    }  
}  
  
fn main() {  
    let c = Closure {env_var: ()};  
    c.call_once();  
}
```

Rust 闭包的实现原理 / 第一类场景

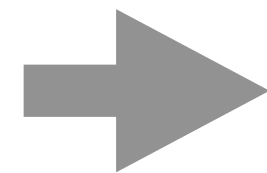
编译器把 FnOnce 的闭包类型看成函数指针

```
fn main() {  
    let c1 = || { "c1"; };  
    let c2 = || { "c2"; };  
    let v = [c1, c2]; // Ok  
  
    let i = "c3";  
    let c3 = || { i };  
    let v = [c1, c2, c3]; // Error  
}
```

Rust 闭包的实现原理 / 第一类场景

编译器把FnOnce 的闭包类型看成函数指针

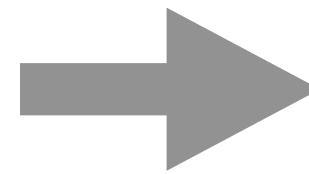
```
fn main() {  
    let c1 = || { "c1" };  
    let c2 = || { "c2" };  
    let v = [c1, c2]; // Ok  
  
    let i = "c3";  
    let c3 = || { i };  
    let v = [c1, c2, c3]; // Error  
}
```



```
error[E0308]: mismatched types  
--> src/main.rs:9:22  
|  
8 |     let c3 = || { i };  
|               ----- the found closure  
9 |     let v = [c1, c2, c3];  
|                       ^^ expected fn pointer, found closure  
|  
= note: expected fn pointer `fn()`  
        found closure `[closure@src/main.rs:8:14: 8:22 i:_]`
```


Rust 闭包的实现原理 / 第二类场景

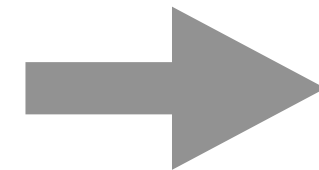
```
fn main() {  
    // 可修改环境变量  
    let mut arr = [1, 2, 3];  
    let mut c2 = |i| {  
        arr[0] = i;  
        println!("{:?}", arr);  
    };  
    c2(0);  
}
```



```
#![feature(unboxed_closures, fn_traits)]  
struct Closure {  
    env_var: [i32; 3],  
}  
  
impl FnOnce<(i32,)> for Closure {  
    type Output = ();  
    extern "rust-call" fn call_once(mut self, args: (i32, )) -> () {  
        self.env_var[0] = args.0;  
        println!("{:?}", self.env_var);  
    }  
}  
  
/**  
    ### 标准库 FnMut trait 的定义  
    pub trait FnMut<Args>: FnOnce<Args> {  
        extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;  
    }  
*/  
impl FnMut<(i32,)> for Closure {  
    extern "rust-call" fn call_mut(&mut self, args: (i32, )) -> () {  
        self.env_var[0] = args.0;  
        println!("{:?}", self.env_var);  
    }  
}  
  
fn main() {  
    let arr = [1,2,3];  
    let mut c = Closure {env_var: arr};  
    // c.call_once( (0,) );  
    c.call_mut( (0,) );  
}
```

Rust 闭包的实现原理 / 第三类场景

```
fn main(){
    let answer = 42;
    let c3 = || {
        println!(
            "The answer to the Ultimate Question of Life, The Universe, and Everything is {}",
            answer
        )
    };
    c3();
}
```



```
#![feature(unboxed_closures, fn_traits)]
struct Closure {
    env_var: i32,
}

impl FnOnce<()> for Closure {
    type Output = ();
    extern "rust-call" fn call_once(self, args: () ) -> () {
        println!(
            "The answer to the Ultimate Question of Life, The Universe, and Everything is {}",
            self.env_var
        )
    }
}

impl FnMut<()> for Closure {
    extern "rust-call" fn call_mut(&mut self, args: () ) -> () {
        println!(
            "The answer to the Ultimate Question of Life, The Universe, and Everything is {}",
            self.env_var
        )
    }
}

/**
    ### 标准库 Fn trait 的定义

    pub trait Fn<Args>: FnMut<Args> {
        extern "rust-call" fn call(&self, args: Args) -> Self::Output;
    }

    */
impl Fn<()> for Closure {
    extern "rust-call" fn call(&self, args: () ) -> () {
        println!(
            "The answer to the Ultimate Question of Life, The Universe, and Everything is {}",
            self.env_var
        )
    }
}

fn main() {
    let answer = 42;
    let mut c = Closure {env_var: answer};
    // c.call_once( () );
    // c.call_mut( () );
    c.call(());
}
```

Rust 闭包的实现原理 / 闭包的类型

1. 如果没有任何捕获变量，则实现 `FnOnce`。
2. 如果有捕获变量，并且会对捕获变量进行修改，则实现 `FnMut`。
3. 如果有捕获变量，并且不会对捕获变量进行修改，则实现 `Fn`。

特殊情况

1. 编译器会把 `FnOnce` 当成 `fn(T)` 函数指针去看待。
2. `Fn`/`FnMut`/`FnOnce` 这三者 trait 的关系是依次 继承，它们正好对应于 所有权语义三件套

小结

1.13 Rust 语法面面观：函数与闭包（下）

闭包的一些其他概念

内容包括：

1. 逃逸闭包与非逃逸闭包
2. 特殊情况：唯一不可变借用
3. 闭包自身实现哪些 trait

逃逸闭包与非逃逸闭包

```
#![feature(unboxed_closures, fn_traits)]  
fn c_mut() -> impl FnMut(i32) -> [i32; 3] {  
    let mut arr = [0, 1, 2];  
    move |i| { arr[0] = i; arr }  
}  
  
fn main() {  
    let i = 42;  
    let mut arr_closure = c_mut();  
  
    // println!("{:?}", arr_closure.call_once((i,)) );  
    println!("{:?}", arr_closure(i));  
}
```



逃逸闭包与非逃逸闭包

```
// Error: FnMut 不能用作逃逸闭包
fn c_mut2() -> impl for<'a> FnMut(&'a str) -> String {
    let mut s = "hello ".to_string();
    move |i|{ s+= i; s}
}

fn main(){
    let i = "world";
    let mut arr_closure = c_mut2(); // Error
}
```

逃逸闭包与非逃逸闭包

```
error[E0507]: cannot move out of `s`, a captured variable in an `FnMut` closure
--> src/main.rs:12:18
   |
11 |     let mut s = "hello ".to_string();
   |           ----- captured outer variable
12 |     |i|{ s+= i; s}
   |           ^ move occurs because `s` has type `String`, which does not implement the
   |           `Copy` trait
```



逃逸闭包与非逃逸闭包

```
#![feature(unboxed_closures, fn_traits)]
fn c_mut() -> impl FnMut(i32) -> [i32; 3] {
    let mut arr = [0, 1, 2];
    move |i|{ arr[0] = i; arr}
}
```

// Error: FnMut 不能用作逃逸闭包

```
fn c_mut2() -> impl for<'a> FnMut(&'a str) -> String {
    let mut s = "hello ".to_string();
    move |i|{ s+= i; s}
}

fn main(){
    let i = "world";
    let mut arr_closure = c_mut2(); // Error
}
```

唯一不可变引用

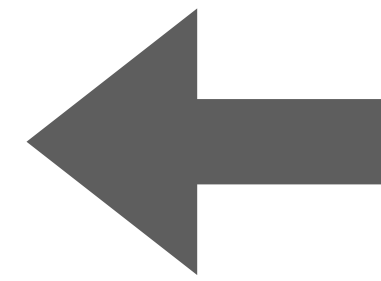
```
fn main(){  
  
    let mut a = [1, 2, 3];  
    let x = &mut a;  
    {  
        let mut c = || { (*x)[0] = 0; };  
        let y = &x; // Error  
        c();  
    }  
    let z = &x; // Ok  
}
```

唯一不可变引用

```
error[E0501]: cannot borrow `x` as immutable because previous closure requires unique access
--> src/main.rs:8:17
|
|
6 |         let mut c = || { (*x)[0] = 0; };
|                        --      - first borrow occurs due to use of `x` in closure
|                        |
|                        closure construction occurs here
7 |         // The following line is an error:
8 |         let y = &x;
|                   ^^ second borrow occurs here
9 |         c();
|         - first borrow later used here
```

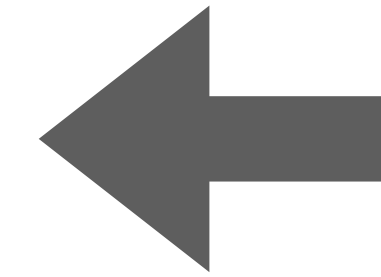
闭包自身实现哪些 trait

1. Sized



所有闭包默认实现

2. Copy / Clone



取决于环境变量的是否实现 Copy
以及它如何被闭包使用的

3. Sync/ Send

闭包自身实现哪些 trait

实现Copy/ Clone 的两条简单规则：

1. 如果环境变量实现了Copy，闭包如果以可变借用方式捕获环境变量，并对其进行修改，则闭包自身不会实现Copy。
2. 如果环境变量自身是Move 语义，则闭包内捕获环境变量的操作涉及修改环境变量或者消耗环境变量，则闭包自身不会实现Copy。

闭包自身实现哪些 trait

```
fn foo<F: Fn() + Copy>(f: F) {  
    f()  
}
```

```
fn main() {  
    let s = "hello".to_owned();  
    let f = || {  
        println!("{}", s);  
    };  
  
    foo(f); // Ok  
}
```

```
fn foo<F: Fn() + Copy>(f: F) {  
    f()  
}
```

```
fn main() {  
    let s = "hello".to_owned();  
    let f = move || {  
        println!("{}", s);  
    };  
  
    foo(f); // Error  
}
```


闭包自身实现哪些 trait

```
error[E0277]: the trait bound `String: Copy` is not satisfied in `[closure@src/main.rs:7:13: 9:6 s:String]`
--> src/main.rs:12:5
   |
1  |     fn foo<F: Fn() + Copy>(f: F) {
   |                               ---- required by this bound in `foo`
   |
...
7  |         let f = move || {
   |         _____-
8  |         |         println!("{}", s);
9  |         |     };
   |         |_____- within this `[closure@src/main.rs:7:13: 9:6 s:String]`
   |
...
12 |         foo(f);
   |         ^^^ within `[closure@src/main.rs:7:13: 9:6 s:String]`, the trait `Copy` is not
   |         implemented for `String`
```

闭包自身实现哪些 trait

实现Sync/Send的三条简单规则：

1. 如果所有捕获变量均实现了Sync，则闭包实现Sync。
2. 如果环境变量都不是「唯一不可变引用」方式捕获的，并且都实现了Sync，则闭包实现Send。
3. 如果环境变量是以「唯一不可变引用」、「可变引用」、Copy或Move 所有权捕获的，那闭包实现Send。

小结

1.14 Rust 语法面面观：模式匹配

内容包括：

1. 模式匹配介绍

2. 模式匹配的位置与模式

模式匹配介绍

模式匹配是一种结构性的解构

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let (a, b) = (1, 2);  
    let Point { x, y } = Point { x: 3, y: 4 };  
  
    assert_eq!(1, a);  
    assert_eq!(2, b);  
    assert_eq!(3, x);  
    assert_eq!(4, y);  
}
```

Rust 中支持模式匹配的位置

1. let 声明
2. 函数和闭包参数
3. match 表达式
4. if let 表达式
5. while let 表达式
6. for 表达式


Rust 中支持模式匹配的位置

/let 声明

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let (a, b) = (1, 2);  
    let Point { x, y } = Point { x: 3, y: 4 };  
  
    assert_eq!(1, a);  
    assert_eq!(2, b);  
    assert_eq!(3, x);  
    assert_eq!(4, y);  
}
```


Rust 中支持模式匹配的位置 / 函数与闭包参数

```
fn sum(x: String, ref y: String) -> String {  
    x + y  
}  
  
fn main() {  
    let s = sum("1".to_owned(), "2".to_owned());  
    assert_eq!(s, "12".to_owned());  
}
```



Rust 中支持模式匹配的位置

/ref 模式

```
fn main() {  
    let a = 42;  
    let ref b = a;  
    let c = &a;  
    assert_eq!(b, c);  
  
    let mut a = [1,2,3];  
    let ref mut b = a;  
    b[0] = 0;  
    assert_eq!(a, [0,2,3]);  
}
```

Rust 中支持模式匹配的位置 /match表达式

```
fn check_optional(opt: Option<i32>) {  
    match opt {  
        Some(p) => println!("has value {}", p),  
        None => println!("has no value"),  
    }  
}  
  
fn handle_result(res: i32) -> Result<i32, dyn Error>{  
    do_something(res)?;  
    // 问号操作符等价于  
    match do_something(res) {  
        Ok(o)  => Ok(o),  
        Err(e) => return SomeError(e),  
    }  
}
```

Rust 中支持模式匹配的位置 /match表达式 演进

```
fn f(x: &Option<String>) {  
    match x {  
        &Some(s) => {println!("{:?}", s)}, // error: error[E0507]: cannot move out of `x.0`  
which is behind a shared reference  
        &None => {println!("nothing")}  
    }  
}  
  
fn main(){  
    let x = Some("hello".to_owned());  
    f(&x);  
}
```

Rust 中支持模式匹配的位置 /match表达式 演进


```
fn f(x: &Option<String>) {  
    match x {  
        &Some(ref s) => {println!("{:?}", s)}, // OK  
        &None => {println!("nothing")}  
    }  
}  
  
fn main(){  
    let x = Some("hello".to_owned());  
    f(&x);  
}
```

Rust 中支持模式匹配的位置 /match表达式 演进

```
fn f(x: &Option<String>) {  
    match *x {  
        Some(ref s) => {println!("{:?}", s)},  
        None => {}  
    }  
}  
  
fn main(){  
    let x = Some("hello".to_owned());  
    f(&x);  
}
```

Rust 中支持模式匹配的位置 /match表达式 演进

```
fn f(x: &Option<String>) {  
    match x {  
        Some(s) => {println!("{:?}", s)},  
        None => {}  
    }  
}  
  
fn main(){  
    let x = Some("hello".to_owned());  
    f(&x);  
}
```



Rust 中支持模式匹配的位置 /切片和动态数组模式

```
fn main() {  
    let arr = [1, 2, 3];  
    match arr {  
        [1, _, _] => "starts with one",  
        [a, b, c] => "starts with something else",  
    };  
  
    // 动态大小数组  
    let v = vec![1, 2, 3];  
    match v[..] {  
        [a, b] => { /* 不匹配 */ }  
        [a, b, c] => { println!("{}", {}, {}, {} ", a, b, c) } // 1, 2, 3  
        _ => { /* 必须包含这条分支, 因为长度是动态的 */ }  
    };  
}
```


Rust 中支持模式匹配的位置

/if let 表达式

```
fn main() {  
    let x: &Option<i32> = &Some(3);  
  
    if let Some(y) = x {  
        y; // &i32  
    }  
}
```

小结

1.15 Rust 语法面面观：智能指针（上）

内容包括：

1. 什么是智能指针？
2. 智能指针的工作机制

什么是智能指针

/指针语义

行为像指针

```
fn main(){  
    let x: Box<i32> = Box::new(42);  
    let y = *x;  
    assert_eq!(y, 42);  
}
```

什么是智能指针

/内存管理机制

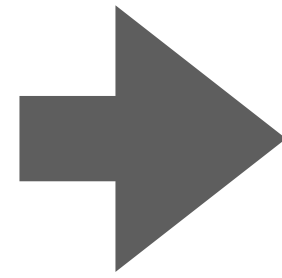
借鉴 Cpp的 RAII

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Box<T> {  
    fn drop(&mut self) {  
        // FIXME: Do nothing, drop is currently performed by compiler.  
    }  
}
```

什么是智能指针

trait 决定了类型的行为

1. Deref trait
2. Drop trait



二者都实现
二选一

智能指针工作机制

/Deref trait

```
pub trait Deref {  
    type Target: ?Sized;  
    fn deref(&self) -> &Self::Target;  
}
```


智能指针工作机制

/Deref trait

```
impl<T: ?Sized> Deref for Box<T> {  
    type Target = T;  
  
    fn deref(&self) -> &T {  
        &**self  
    }  
}
```

智能指针工作机制

/ 自定义智能指针

```
use std::ops::Deref;
struct MySmartPointer<T>(T);
impl<T> MySmartPointer<T> {
    fn new(x: T) -> MySmartPointer<T> {
        MySmartPointer(x)
    }
}

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

    fn deref(&self) -> &T {
        &self.0
    }
}

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

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

小结

1.16 Rust 语法面面观：智能指针（下）

内容包括：

1. 智能指针智能在何处
2. Rust 标准库包含了哪些指针

智能在何处？

1. 可以自动解引用，提升开发体验
2. 可以自动化管理内存，安全无忧

智能在何处? / 自动解引用 - 点调用操作

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

struct MySmartPointer<T>(T);

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

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

    fn deref(&self) -> &T {
        &self.0
    }
}
```

```
struct User {
    name: &'static str
}

impl User {
    fn name(&self) {
        println!("{:?}", self.name);
    }
}

fn main() {
    let u = User{name: "Alex"};
    let y = MySmartPointer::new(u);


    y.name();
}
```

智能在何处? / 自动解引用 - 函数参数

```
fn takes_str(s: &str) {  
    println!("{:?}", s); // "Hello"  
}  
  
fn main() {  
    let s = String::from("Hello");  
    takes_str(&s);  
}
```


智能在何处? / 自动解引用 - 函数参数

```
impl ops::Deref for String {  
    type Target = str;  
  
    #[inline]  
    fn deref(&self) -> &str {  
        unsafe { str::from_utf8_unchecked(&self.vec) }  
    }  
}
```



智能在何处？ / 自动解引用 - 需要注意的地方

1. 使用 `*x` 这样手工解引用的方式，等价于 `*(x.deref())`
2. 使用 点调用或在函数参数位置上对x自动解引用则是等价于 `x.deref()`

标准库中的智能指针

1. Box<T>

2. Vec<T> 和 String

3. Rc<T> 和 Arc<T>

4. HashMap<K, V>

标准库中的智能指针

标准库中的实现

```
impl<T: ?Sized> Deref for &T {  
    type Target = T;  
  
    fn deref(&self) -> &T {  
        *self  
    }  
}  
  
impl<T: ?Sized> !DerefMut for &T {}  
  
impl<T: ?Sized> Deref for &mut T {  
    type Target = T;  
  
    fn deref(&self) -> &T {  
        *self  
    }  
}
```

小结

1.17&1.18 Rust 语法面面观：字符与字符串

内容包括：

1. Rust 中字符和字符串的特点
2. Rust 标准库导读

字符

1. Unicode 标量值，其值对应于 Rust 中 u32 类型
2. 占 4 个字节
3. 可方便地转换为 utf8 编码字节序列

字符

```
1 ▼ fn main() {  
2     let tao = '道';  
3     let tao_u32 = tao as u32;  
4     assert_eq!(36947, tao_u32);  
5     println!("U+{:x}", tao_u32); // U+9053  
6     println!("{}", tao.escape_unicode()); // \u{9053}  
7     assert_eq!(char::from(65), 'A');  
8     assert_eq!(std::char::from_u32(0x9053), Some('道'));  
9     assert_eq!(std::char::from_u32(36947), Some('道'));  
10    // 该数字并不是一个有效的Unicode标量值  
11    assert_eq!(std::char::from_u32(12901010101), None);  
12 }
```

字符串

1. UTF-8 字节序列, “Vec<u8>”
2. str 和 String 两大常用字符串类型
3. 其他 字符串分类:
 - Cstr/Cstring
 - OsStr/OsString
 - Path/PathBuf

字符串

```
1 use std::str;
2 fn main() {
3     // 将UTF-8序列转为字符串
4     let tao = str::from_utf8(&[0xE9u8, 0x81u8, 0x93u8]).unwrap();
5     assert_eq!("道", tao);
6     // 将16进制Unicode码位转为字符串
7     assert_eq!("道", String::from("\u{9053}"));
8     let unicode_x = 0x9053;
9     let utf_x_hex = 0xe98193;
10    let utf_x_bin = 0b1110100110000000110010011;
11    println!("unicode_x: {:b}", unicode_x);
12    println!("utf_x_hex: {:b}", utf_x_hex);
13    println!("utf_x_bin: 0x{:x}", utf_x_bin);
14 }
```

Rust 标准库导读：字符与字符串相关

Rust 标准库导读：字符与字符串相关

1. 查看类型自身的介绍
2. 查看类型自身实现的方法
3. 查看类型实现的 trait

小结

作业

自己再阅读一遍文档，对字符和字符串中的方法做一个分类总结。

1.19&1.20 Rust 语法面面观：集合容器

内容包括：

1. Rust 中集合容器的分类
2. Rust 标准库导读
3. Rust 标准库集合容器为什么没有统一的接口（trait）？

集合容器

1. Vec<T>, 动态可增长数组
2. VecDeque<T>, 基于环形缓冲区的先进先出双端队列
3. LinkedList<T>, 非侵入式双向链表实现
4. BinaryHeap<T>, 二叉堆实现, 可用做优先队列
5. HashMap<K,V> / BTreeMap<K,V>
6. HashSet<T> / BTreeMap<T>

Rust 标准库导读：集合容器

Rust 标准库导读：集合容器

1. `Vec<T>`，动态可增长数组
2. `LinkedList<T>`，非侵入式双向链表实现
3. `HashMap<K,V>`

Rust 中集合容器为什么没有统一的接口 (trait)

需要 GAT

(Generic Associated Type)

```
1 trait Collection<T> {  
2     fn empty() -> Self;  
3     fn add(&mut self, value: T);  
4  
5     type Iter<'iter>: Iterator<Item=T>;  
6     // Here, we use associated type constructors:  
7     fn iterate<'iter>(&'iter self) -> Self::Iter<'iter>;  
8 }  
9  
10 impl<T> Collection<T> for List<T> {  
11     type Iter = ListIter<'iter, T>;  
12  
13     fn empty() -> List<T> {  
14         List::new()  
15     }  
16  
17     fn add(&mut self, value: T) {  
18         self.prepend(value);  
19     }  
20  
21     fn iterate<'iter>(&'iter self) -> ListIter<'iter, T> {  
22         self.iter()  
23     }  
24 }
```

小结

作业

自己再阅读一遍文档，总结集合的 API 设计和实现有什么特点？
看能否找出一致性？

1.21&1.22 Rust 语法面面观：迭代器

内容包括：

1. 迭代器模式
2. 标准库导读
3. 第三方库：itertools

迭代器模式

1. 设计模式中的一种行为模式
2. 常与集合使用，不暴露集合底层的情况下遍历集合元素
3. 将集合的遍历行为抽象为单独的迭代对象

迭代器模式

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

for 循环

外部迭代器语法糖

```
1 ▼ fn main() {  
2     let v = vec![1, 2, 3, 4, 5];  
3 ▼ { // 等价于for循环的scope  
4     let mut _iterator = v.into_iter();  
5 ▼     loop {  
6 ▼         match _iterator.next() {  
7 ▼             Some(i) => {  
8                 println!("{}", i);  
9                 }  
10             None => break,  
11         }  
12     }  
13 }  
14 }
```

自定义内部迭代器

```
1 ▼ trait InIterator<T: Copy> {  
2     fn each<F: Fn(T) -> T>(&mut self, f: F);  
3 }  
4 ▼ impl<T: Copy> InIterator<T> for Vec<T> {  
5     fn each<F: Fn(T) -> T>(&mut self, f: F) {  
6         let mut i = 0;  
7         while i < self.len() {  
8             self[i] = f(self[i]);  
9             i += 1;  
10        }  
11    }  
12 }  
13  
14 ▼ fn main(){  
15     let mut v = vec![1,2,3];  
16     v.each(|i| i * 3);  
17     assert_eq!([3, 6, 9], &v[..3]);  
18 }
```

Rust 标准库导读：迭代器

Rust 标准库导读：迭代器

1. Iterator、Iter、IterMut
2. 标准库内建的迭代器适配器
3. 标准库内建的迭代器消费者
4. 迭代器在字符串和集合中的应用

第三方库：Itertools

小结

作业

1. 自己再阅读一遍文档，总结迭代器的 API 设计和实现有什么特点？
2. 使用 Itertools 编写一些示例代码，并阅读感兴趣的方法源码



扫码试看/订阅

《张汉东的Rust实战课》视频课程