Rust第二次学习

河南科技大学-徐堃元

2023/10/13

本节课程内容

- 枚举与Options
- Rust 的包、create 和模块
- Rust 泛型和 trait
- 生命周期简介

《Rust程序设计语言》：

https://kaisery.github.io/trpl-zh-cn/

《通过例子学Rust》：

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

Rust语言中文社区：

https://rustcc.cn/

枚举（enum）

枚举定义

```
enum IpAddrKind {
    V4,
    V6,
}
```

包含成员关联类型的定义

```
enum IpAddr {
        V4(String),
        V6(String),
    }
```

创建成员实例

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

枚举实例


```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

成员初始化

```
let messages = [
        Message::Move { x: 10, y: 30 },
        Message::Echo(String::from("hello world")),
        Message::ChangeColor(200, 255, 255),
        Message::Quit,
    ];
```

Option枚举

`Option` 是标准库定义的另一个枚举。`Option` 类型应用广泛因为它编码了一个非常普遍的场景，即一个值要么有值要么没值。

```
enum Option<T> {
    None,
    Some(T),
}
```

Option值

```
let some_number = Some(5);
let some_char = Some('e');

let absent_number: Option<i32> = None;
```

match控制流

Rust 有一个叫做 `match` 的极为强大的控制流运算符，它允许我们将一个值与一系列的模式相比较，并根据相匹配的模式执行相应代码。

类似其他语言的 switch

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

match控制流

在match分支中运行多行代码

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

分支中绑定匹配模式值

```rust
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

匹配 Option

```
fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
```

通配符匹配

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

使用'_'通配符

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

match控制流

match 控制流结构

- match 是 Rust 的一个强大的控制流运算符，它可以将一个值与一系列的模式相比较，并根据匹配的模式执行相应的代码。
- match 的分支由一个模式和一些代码组成，可以使用变量或通配符来绑定匹配的模式的部分值。
- match 是穷尽的，必须覆盖所有可能的情况，否则会编译错误。

`Option<T>` 和 match

- `Option<T>` 是 Rust 的一个枚举，它表示一个值可能存在或不存在。
- match 可以用来处理 `Option<T>` 的成员，Some(T) 或 None，并根据其中是否有值执行不同的操作。
- 使用 _ 模式可以匹配任意值而不绑定到该值，用于忽略不需要使用的值或满足穷尽性要求。

if let 简洁控制流

`if let` 语法让我们以一种不那么冗长的方式结合 `if` 和 `let`，来处理只匹配一个模式的值而忽略其他模式的情况。

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

使用if let

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

Rust模块

这里有一个需要说明的概念 “作用域（scope）”：代码所在的嵌套上下文有一组定义为 “in scope” 的名称。当阅读、编写和编译代码时，程序员和编译器需要知道特定位置的特定名称是否引用了变量、函数、结构体、枚举、模块、常量或者其他有意义的项。你可以创建作用域，以及改变哪些名称在作用域内还是作用域外。同一个作用域内不能拥有两个相同名称的项；可以使用一些工具来解决名称冲突。

Rust 有许多功能可以让你管理代码的组织，包括哪些内容可以被公开，哪些内容作为私有部分，以及程序每个作用域中的名字。这些功能。这有时被称为 “模块系统（the module system）”，包括：

- **包**（*Packages*）：Cargo 的一个功能，它允许你构建、测试和分享 crate。
- **Crates**：一个模块的树形结构，它形成了库或二进制项目。
- **模块**（*Modules*）和 **use**：允许你控制作用域和路径的私有性。
- **路径**（*path*）：一个命名例如结构体、函数或模块等项的方式

包和 create

包和 crate

- 包是提供一系列功能的一个或者多个 crate，包含一个 Cargo.toml 文件。
- crate 是 Rust 在编译时最小的代码单位，有两种形式：二进制项和库。
- crate root 是一个源文件，是 crate 的根模块。

crate 的约定

- src/main.rs 是一个与包同名的二进制 crate 的 crate 根。
- src/lib.rs 是一个与包同名的库 crate 的 crate 根。
- src/bin 目录下的每个文件都会被编译成一个独立的二进制 crate。

模块系统与模块树

模块系统

- 模块可以将crate中的代码进行分组和封装，提高可读性和重用性。
- 路径是一种命名项的方式，可以使用绝对路径或相对路径。
- use关键字可以在一个作用域内创建一个项的快捷方式，减少长路径的重复。
- pub关键字可以将模块或模块内的项标记为公开的，使外部代码可以使用它们。

模块树

- crate根文件（src/lib.rs或src/main.rs）是crate模块结构的根，也是名为crate的隐式模块的根。
- 在crate根文件中，可以声明新的模块，使用mod关键字和花括号或分号。
- 在其他文件中，可以定义子模块，使用mod关键字和花括号或分号，并在以父模块命名的目录中寻找子模块代码。
- 模块树可以用来展示crate中的模块层次结构，以及模块之间的父子和兄弟关系。

引用模块的路径

- **绝对路径**（*absolute path*）是以 crate 根（root）开头的全路径；对于外部 crate 的代码，是以 crate 名开头的绝对路径，对于当前 crate 的代码，则以字面值 `crate` 开头。
- **相对路径**（*relative path*）从当前模块开始，以 `self`、`super` 或当前模块的标识符开头。

绝对路径和相对路径都后跟一个或多个由双冒号（`::`）分割的标识符。

声明模块

一个crate中，可以包含多个模块。

声明的模块将通过以下几种途径去寻找：

- 内联，在大括号中，当mod xxx后方不是一个分号而是一个大括号
- 在文件 src/xxx.rs
- 在文件 src/xxx/mod.rs

声明子模块

在除了根节点之外的其他文件的模块中，也可以声明子模块。

例如，在src/xxx.rs中定义了一个mod yyy。那么在以下几个位置寻找：

- 内联，在大括号中，当mod yyy后方不是一个分号而是一个大括号
- 在文件 src/xxx/yyy.rs
- 在文件 src/xxx/yyy/mod.rs

使用use关键字引用模块

例如在xxx模块的yyy子模块下面定义了一个zzz

那么可以通过crate::xxx::yyy::zzz;来使用它。

使用use关键字

使用use crate::xxx::yyy::zzz;之后，之后可以直接用zzz

使用as关键字

（这和python中import xxx as yyy类似）

use crate::xxx::yyy::zzz as z;

模块私有和公有

在rust中，默认对父模块来说，所有内容都是私有的。

所以在mod关键词之前加pub，让父模块可以访问子模块。

子模块的函数或者变量，也要加pub，让父模块访问。

一言以蔽之，想要访问私有，就用pub。

Rust泛型

泛型数据类型

- 泛型数据类型可以让代码适用于多种不同的具体类型，避免重复和冗余。
- 泛型可以用在函数签名、结构体、枚举和方法中，用尖括号 <> 来声明泛型参数的名称。
- 泛型参数可以有限制（constraint），指定它们必须实现某些 trait 或具有某些行为。

单态化

- Rust 通过在编译时进行泛型代码的单态化，将泛型参数替换为具体类型，来保证运行时的效率。
- 单态化可以避免使用泛型带来的性能损失，使得泛型代码的执行速度与手写的具体类型代码一样快。

在函数定义中使用泛型

```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);
    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
```

在函数定义中使用泛型

```rust
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

结构体定义中的泛型

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

多个泛型类型

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

方法定义中的泛型

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Rust trait

Trait 定义和实现

- trait 定义了某个特定类型拥有可能与其他类型共享的功能。
- 可以使用 impl 关键字为类型实现 trait，也可以为 trait 提供默认的行为。
- 可以使用 impl Trait 或泛型参数加 trait bound 的语法来指定函数参数或返回值是实现了某个 trait 的类型。

Trait 作为参数和返回值

- 使用 impl Trait 或 trait bound 可以让函数接受多种不同类型的参数，只要它们实现了指定的 trait。
- 使用 impl Trait 作为返回值可以让函数返回一个只有编译器知道的类型，只要它实现了指定的 trait。
- 不能使用 impl Trait 来返回多种不同类型，需要使用 trait 对象来实现这个功能。

有条件地实现方法和 trait

- 可以使用带有 trait bound 的泛型参数的 impl 块，来有条件地只为那些实现了特定 trait 的类型实现方法。
- 可以对任何满足特定 trait bound 的类型有条件地实现 trait，这被称为 blanket implementations。

定义 trait

一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。trait 定义是一种将方法签名组合起来的方法，目的是定义一个实现某些目的所必需的行为的集合。

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

为 trait增加默认实现

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

为类型实现 trait


```rust
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}
impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

使用 trait方法


```
use aggregator::{Summary, Tweet};

fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

使用 trait 的默认实现

```
let article = NewsArticle {
    headline: String::from("Penguins win the Stanley Cup Championship!"),
    location: String::from("Pittsburgh, PA, USA"),
    author: String::from("Iceburgh"),
    content: String::from(
        "The Pittsburgh Penguins once again are the best \
         hockey team in the NHL.",
    ),
};

println!("New article available! {}", article.summarize());
```

这段代码会打印 `New article available! (Read more...)`。

多个 trait 方法

默认实现允许调用相同 trait 中的其他方法，哪怕这些方法没有默认实现。如此，trait 可以提供很多有用的功能而只需要实现指定一小部分内容。

```rust
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

对应定义

```rust
impl Summary for Tweet {
    fn summarize_author(&self) -> String {
        format!("@{}", self.username)
    }
}
```

基于默认实现使用多个 trait 方法

```
let tweet = Tweet {
    username: String::from("horse_ebooks"),
    content: String::from(
        "of course, as you probably already know, people",
    ),
    reply: false,
    retweet: false,
};

println!("1 new tweet: {}", tweet.summarize());
```

这会打印出 `1 new tweet: (Read more from @horse_ebooks...)`。

trait 作为参数

```rust
pub fn notify(item: &impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

trait bound语法

```rust
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

获取多个参数

```rust
pub fn notify(item1: &impl Summary, item2: &impl Summary) {}

pub fn notify<T: Summary>(item1: &T, item2: &T) {}
```

使用 + 指定多个 trait bound

如果 `notify` 需要显示 `item` 的格式化形式，同时也要使用 `summarize` 方法，那么 `item` 就需要同时实现两个不同的 trait：`Display` 和 `Summary`。这可以通过 `+` 语法实现：

```
pub fn notify(item: &(impl Summary + Display)) {
```

`+` 语法也适用于泛型的 trait bound：

```
pub fn notify<T: Summary + Display>(item: &T) {
```

通过指定这两个 trait bound，`notify` 的函数体可以调用 `summarize` 并使用 `{}` 来格式化 `item`。

使用 where 简化 trait bound

然而，使用过多的 trait bound 也有缺点。每个泛型有其自己的 trait bound，所以有多个泛型参数的函数在名称和参数列表之间会有很长的 trait bound 信息，这使得函数签名难以阅读。为此，Rust 有另一个在函数签名之后的 `where` 从句中指定 trait bound 的语法。所以除了这么写：

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

还可以像这样使用 `where` 从句：

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{
```

返回实现了 trait 的类型


```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

使用 trait bound 有条件地实现方法

```rust
use std::fmt::Display;

struct Pair<T> {
    x: T,
    y: T,
}

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

生命周期

生命周期是另一类我们已经使用过的泛型。不同于确保类型有期望的行为，生命周期确保引用如预期一直有效。

生命周期的主要目标是避免**悬垂引用**（*dangling references*），后者会导致程序引用了非预期引用的数据。

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

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

函数中的泛型生命周期

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

这段代码无法通过编译

生命周期注解语法

生命周期注解并不改变任何引用的生命周期的长短。相反它们描述了多个引用生命周期相互的关系，而不影响其生命周期。与当函数签名中指定了泛型类型参数后就可以接受任何类型一样，当指定了泛型生命周期后函数也能接受任何生命周期的引用。

```
&i32        // 引用
&'a i32     // 带有显式生命周期的引用
&'a mut i32 // 带有显式生命周期的可变引用
```

单个的生命周期注解本身没有多少意义，因为生命周期注解告诉 Rust 多个引用的泛型生命周期参数如何相互联系的。

函数签名中的生命周期注解

为了在函数签名中使用生命周期注解，需要在函数名和参数列表间的尖括号中声明泛型生命周期（*lifetime*）参数，就像泛型类型（*type*）参数一样。

我们希望函数签名表达如下限制：也就是这两个参数和返回的引用存活的一样久。（两个）参数和返回的引用的生命周期是相关的。

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

现在函数签名表明对于某些生命周期 `'a`，函数会获取两个参数，它们都是与生命周期 `'a` 存在的一样长的字符串 slice。函数会返回一个同样也与生命周期 `'a` 存在的一样长的字符串 slice。它的实际含义是 `longest` 函数返回的引用的生命周期与函数参数所引用的值的生命周期的较小者一致。

当具体的引用被传递给 `longest` 时，被 `'a` 所替代的具体生命周期是 `x` 的作用域与 `y` 的作用域相重叠的那一部分。换一种说法就是泛型生命周期 `'a` 的具体生命周期等同于 `x` 和 `y` 的生命周期中较小的那一个。

结构体定义中的生命周期注释


```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.').next().expect("Could not find a '.'");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

生命周期省略


```rust
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

生命周期省略

函数或方法的参数的生命周期被称为 **输入生命周期**（*input lifetimes*），而返回值的生命周期被称为 **输出生命周期**（*output lifetimes*）。

编译器采用三条规则来判断引用何时不需要明确的注解。第一条规则适用于输入生命周期，后两条规则适用于输出生命周期。如果编译器检查完这三条规则后仍然存在没有计算出生命周期的引用，编译器将会停止并生成错误。这些规则适用于 `fn` 定义，以及 `impl` 块。

第一条规则是编译器为每一个引用参数都分配一个生命周期参数。换句话说就是，函数有一个引用参数的就有一个生命周期参数：`fn foo<'a>(x: &'a i32)`，有两个引用参数的函数就有两个不同的生命周期参数，`fn foo<'a, 'b>(x: &'a i32, y: &'b i32)`，依此类推。

第二条规则是如果只有一个输入生命周期参数，那么它被赋予所有输出生命周期参数：`fn foo<'a>(x: &'a i32) -> &'a i32`。

第三条规则是如果方法有多个输入生命周期参数并且其中一个参数是 `&self` 或 `&mut self`，说明是个对象的方法 (method)，那么所有输出生命周期参数被赋予 `self` 的生命周期。第三条规则使得方法更容易读写，因为只需更少的符号。

方法定义中的生命周期注释

```
impl<'a> ImportantExcerpt<'a> {
    fn level(&self) -> i32 {
        3
    }
}
```

`impl`之后和类型名称之后的生命周期参数是必要的，不过因为第一条生命周期规则我们并不必须标注`self`引用的生命周期。

```
impl<'a> ImportantExcerpt<'a> {
    fn announce_and_return_part(&self, announcement: &str) -> &str {
        println!("Attention please: {}", announcement);
        self.part
    }
}
```

静态生命周期

一个值的生命周期贯穿整个进程的生命周期。

那么其引用的生命周期也具有静态生命周期。

例如可以用'static标志标识一个静态生命周期的字符串引用：

```
let s: &'static str = "hello world";
```

以下是一些具有静态生命周期的：

字符串字面量

全局变量

静态变量

使用了Box::leak之后的堆内存。

动态生命周期

分配在堆和栈上面的都具有动态生命周期

Rust中，堆内存的生命周期会对应的栈内存的生命周期绑定在一起。

泛型类型参数的生命周期标注


```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

对应rustlings练习

modules

options

errors

generics

traits

lifetime

tests1~4