TinTin

Office Hour 5

讲师：Mike Tang
助教：雷昕鑫

TinTin

生命周期、Option、Result与错误处理、宏

- Lifetime和Scope：在Rust中，每个引用都有一个生命周期，这是引用保持有效的范围。生命周期的主要目标是防止悬挂引用和空指针。
- Option：Option是一个枚举，它表示一个值可能存在（Some）或者不存在（None）。这是Rust处理null或者nil的一种方式，它可以帮助避免空指针异常。
- Result与错误处理：Result是一个枚举，它表示一个操作可能成功（Ok）或者失败（Err）。在Rust中，错误处理通常通过返回Result类型来进行，这样可以确保错误不会被忽略，并且可以在编译时检查。
- 宏（Macros）：宏是一种在编译时执行代码并生成其他代码的方式。它们可以用来减少重复代码，创建DSL（领域特定语言），或者实现元编程等功能。

TinTin

Lifetime和Scope

作用域（Scope）：在Rust中，作用域是一个变量有效的范围，也就是从变量被声明开始，到它被销毁的地方。在作用域之外，你不能访问这个变量。例如：

```
{
    let s = "hello"; // s的作用域从这里开始
    // 在这里可以访问s
} // s的作用域在这里结束，s被销毁
// 在这里不能再访问s
```

Lifetime和Scope

生命周期（Lifetime）：生命周期是Rust中的一个抽象概念，它代表了一段程序中的一段时间。在Rust中，每个引用都有一个生命周期，这是引用保持有效的范围。生命周期的主要目标是防止悬挂引用和空指针。

生命周期参数并不改变实际的生命周期，它们只是编译器用来检查引用安全性的工具。实际的生命周期是由变量的作用域决定的，而不是生命周期参数。

生命周期有个特点，它总是由编译器来推断的，你无法自己给引用分配生命周期，开发者只能通过标记生命周期给编译器检查和证明。

TinTin

Lifetime和Scope

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

在这个例子中，'a是一个生命周期参数，它告诉Rust x、y和返回值的引用都必须有相同的生命周期。这样，Rust编译器就可以确保在函数返回的引用仍然有效时，x和y仍然没有被销毁。

TinTin

Lifetime和Scope

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

这个函数的签名中，'b: 'a表示'b的生命周期至少和'a一样长。因此，无论x和y哪个字符串更长，函数都可以安全地返回一个生命周期为'a的引用。

Lifetime和Scope

结构体中的生命周期：当你在结构体中使用引用时，你需要指定这个引用的生命周期。例如：

```
struct Person<'a> {
    name: &'a str,
}
```

在这个例子中，'a是一个生命周期参数，它意味着`name`字段的引用必须在Person实例的整个生命周期中都是有效的。

TinTin

Lifetime和Scope

实现方法或trait时的生命周期：当你在方法或trait的实现中使用引用时，你需要指定这些引用的生命周期。例如：

```
impl<'a> Person<'a> {
    fn greet(&self) {
        println!("Hello, my name is {}", self.name);
    }
}
```

在这个例子中，'a是一个生命周期参数，它告诉Rust self的引用必须有这个生命周期。

Lifetime和Scope

静态生命周期：'static生命周期是一个特殊的生命周期，它表示引用的整个程序期间都是有效的。例如：

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

在这个例子中，s的生命周期是'static，这意味着s在整个程序期间都是有效的。

TinTin

Option<T>

Option<T> 是 Rust 中的一个枚举类型，它的产生是为了解决空值（null）问题。在许多其他编程语言中，null 值经常导致运行时错误，因为程序员往往忘记或者不知道需要检查变量是否为 null。Rust 通过 Option<T> 类型来确保程序员必须处理可能的空值，从而避免了这类错误。

TinTin

Option<T> 匹配

1.使用 match 语句：match 语句可以用来匹配 Option<T> 的两种可能的变体，并对每种情况进行处理。例如：

```
let optional = Some(5);
match optional {
    Some(i) => println!("Value is: {}", i),
    None => println!("No value"),
}
```

TinTin

Option<T> 匹配

2.使用 if let 语句：if let 语句是一种更简洁的方式来处理 Option<T>。它只处理 Some(T) 的情况，如果是 None，则不执行任何操作。例如：

```
let optional = Some(5);
if let Some(i) = optional {
    println!("Value is: {}", i);
} else {
    println!("Value is None");
}
```

TinTin

Option<T> 解包

1.expect(message: &str) –> T：这个方法返回 Option<T> 中的 T 值。如果 Option 是 None，它会 panic 并显示提供的错误消息。
2.unwrap() –> T：这个方法也返回 Option<T> 中的 T 值。如果 Option 是 None，它也会 panic，但不会显示特定的错误消息。

3.unwrap_or(default: T) –> T：这个方法返回 Option<T> 中的 T 值，如果 Option 是 None，它会返回提供的默认值。

4.unwrap_or_else(f: F) –> T：这个方法返回 Option<T> 中的 T 值，如果 Option 是 None，它会调用提供的函数 f 并返回其结果。这个函数通常用于在返回默认值时进行一些计算。

TinTin

Option<T> 链式处理

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=31023a015a1d33f76e6f5310adb8a46b

TinTin

Result<T, E>

在编程中，错误处理是一个重要的部分。在很多语言中，错误通常通过异常来处理，但是在 Rust 中，异常并不常用。相反，Rust 提供了一个叫做 Result<T, E> 的枚举类型来处理可能会失败的操作。这是 Rust 的一种错误处理模式，它使得错误处理更加显式和直接。

TinTin

Result<T, E>

错误可以通过显式的match语句进行处理，这样的代码往往冗长且难以阅读。为了解决这个问题，Rust引入了?运算符，它可以在函数返回Result类型时，自动处理Err情况，使得错误处理变得更加简洁和直观。

TinTin

宏

Rust的宏（macro）是一种元编程工具，它们在编译时期运行并生成代码。

Rust的宏有两种形式：声明式宏（macro_rules!）和过程宏（包括派生宏、属性宏和函数宏）。

宏的主要作用是减少代码重复，提高代码的可读性和可维护性。

TinTin

宏

属性

https://cheats.rs/#tooling-directives
https://cheats.rs.kingfree.moe/#tooling-directives

I 课程关键内容回顾

TinTin

宏

```
macro_rules! add{
    ($a:expr,$b:expr)=>{
        {
            $a+$b
        }
    };
    ($a:expr)=>{
        {
            $a
        }
    }
}
```

```
fn main(){
    let x = 0;
    println!("{}", add![1,2]);
    println!("{}", add!(x));
    println!("{}", add!{1,3});
}
```

在Rust中，宏调用可以使用不同的分隔符，包括圆括号()、方括号[]和大括号{}。这是Rust语言设计的一部分，主要是为了提供更多的灵活性，让你可以根据不同的上下文和风格选择最适合的分隔符。

例如，如果你的宏看起来更像一个函数调用，你可能会选择使用圆括号。如果你的宏看起来更像一个类型或者表达式，你可能会选择使用方括号。如果你的宏包含了一大块代码，你可能会选择使用大括号。

I 作业点评

TinTin

请搜索相关文档，实现：一个简单的声明宏
并理解其代码结构，和编译过程。

优秀作业展示

dwoura同学

```rust
//简易vector
#[macro_export]
macro_rules! vector {
    ($($x:expr),*) => {
        {
            let mut temp_vec = Vec::new();
            $(temp_vec.push($x);)*
            temp_vec
        }
    };
}

fn main() {
    //区别原来的vec![]。
    let v = vector![1, 2, 3, 4];
    for i in v.iter() {
        println!("{}", i);
    }
}
```

- 代码结构
  - `#[macro_export]` 注释将宏进行了导出，这样其他包才能引入该宏。
  - `macro_rule!` 宏定义，定义的时候不需要！，只有调用的时候才需要。
  - `( $( $x:expr ),* )` 模式解析。其中参数们先被一对 ( ) 括起来，小括号里写自定义的参数 $(),然后 `*` 说明 `*` 之前的模式会被匹配零次或任意多次。
  - `$x:expr` 的 `expr` 表示rust中任意表达式，然后给出一个新模式叫 `$x`。由于可以匹配任意表达式， 所以 `$x` 模式既可以跟整数 `1` 进行匹配，也可以跟字符串 "hello" 进行匹配: `vec!["hello", "world"]`。（好像有点泛型的感觉）
  - `=>{}` 里的 `$(temp_vec.push($x);)*` 这句也是遵从了 `$()` 中写表达式，并且 `*` 有任意多个含有不同 `$x` 的 `$(temp_vec.push($x);)`

  当调用vector![1,2,3,4]时:

```rust
{
    let mut temp_vec = Vec::new();
    $(temp_vec.push($x);)*
    temp_vec
}
//等同于
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

- 编译过程

main函数中调用vector![1, 2, 3, 4]时，rust编译器会解析vector宏并对其中整体元素进行模式匹配，匹配成功后就会把宏内部的代码片段替换为相应代码，最后执行相应代码并返回结果。

TinTin

- 视频推荐

- 喜欢历史的程序君
- Rust Axum Full Course - Web Development

- 网站

- Rust On Nails

Q&A

TinTin

答疑讨论

THANKS

Twitter | YouTube | Discord