浅谈 Rust Ownership

Cdot CTO John Wu

cdot network

一块链习

栈和堆

- 栈上的数据：静态大小（整数类）
- 堆上的数据：动态大小，动态分配。需要小心处理，从来都是个棘手的问题（String）

处理堆上数据的方法

- GC (Java)
- 自行 (de)allocate (C)
- Ownership (Rust)

Ownership 的几条基本规则

- 每个 rust 中的值都有一个 owner （是一个变量）
- 同时最多只有一个 owner
- 当 owner 离开作用域之后，对应的值就会被清理（ drop ）（看起来没啥区别？和生存周期有关）

```
fn main() {
    let a = "a".to_string();
    {
    let b = "b".to_string();
    }
    println!("{}", a);
    // error!
    // println!("{}", b);
}
```

Move 语义 1

```
fn main() {
      let a = "abc".to_string()
      let b = a;
      println!("{}", a);
}
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:4:22
  |
2 |     let a = "abc".to_string();
  |         - move occurs because `a` has type `std::string::String`, which does not implement the `Copy` trait
3 |     let b = a;
  |             - value moved here
4 |     println!("{}", a);
  |                    ^ value borrowed here after move
```

Move 语义 1

```
fn main() {
        let a = "abc".to_string()
        let b = a.clone();
        println!("{}", a);
}
```

Move 语义 1

```
fn main() {
        let a = "123";
        let b = a;
        println!("{}", a);
}
```

Move 语义 2

```
fn do_nothing(_a: String) {}
fn main() {
    let a = "a".to_string();
    do_nothing(a);
    println!("{}", a);
}
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:5:20
  |
3 |     let a = "a".to_string();
  |         - move occurs because `a` has type `std::string::String`, which does not implement the `Copy` trait
4 |     do_nothing(a);
  |                - value moved here
5 |     println!("{}", a);
  |                    ^ value borrowed here after move
```

Move 语义 3

```
fn do_something(a: String) -> String {
    a
}
fn main() {
    let a = "a".to_string();
    let b = do_something(a);
    println!("{}", a);
}
```

关于引用（reference）

- 可变引用 &mut 和不可变引用 &
- 可以有 1 个可变引用 &mut
- 可以有多个不可变引用 &
- 两者不能并存

解决的问题 1

```
fn main() {
      let mut a = "a".to_string();
      let b = &a;
      let mut c = &mut a;
      c.push('c’);
      println!("{}", c);
      println!(“{}”, b); //  考虑程序很大的情况
}
error[E0502]: cannot borrow `a` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:17
  |
3 |     let b = &a;
  |             -- immutable borrow occurs here
4 |     let mut c = &mut a;
  |                 ^^^^^^ mutable borrow occurs here
...
7 |     println!("{}", b);
  |                    - immutable borrow later used here
```

解决的问题 2

```
fn main() {
      let mut input = vec![1, 2, 3];
      for i in input {
            input.push(1);
      }
}
error[E0382]: borrow of moved value: `input`
 --> src/main.rs:4:9
  |
2 |     let mut input = vec![1, 2, 3];
  |         --------- move occurs because `input` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
3 |     for i in input {
  |              ----- value moved here
4 |         input.push(1);
  |         ^^^^^ value borrowed here after move
```

解决的问题 2

```
fn main() {
    let mut input = vec![1, 2, 3];
    for i in &input {
        input.push(1);
    }
}
3 |     for i in &input {
  |              ------
  |              |
  |              immutable borrow occurs here
  |              immutable borrow later used here
4 |         input.push(1);
  |         ^^^^^^^^^^^^^ mutable borrow occurs here
```

解决的问题 2

```
fn main() {
    let mut input = vec![1, 2, 3];
    for i in input.clone() {
        input.push(1);
    }
}
```

特殊生存周期

```
error[E0106]: missing lifetime specifier
--> src/main.rs:3:17
|
3 | fn teststr() -> &str {
| ^ help: consider giving it a 'static lifetime: `&'static`
|
= help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from
fn teststr() -> &str {
    let a = "foobar";
    println!("In function: {}", a);
    a
    }
```

特殊生存周期

```
fn fn1() -> &'static str {
    let a = "foobar";
    println!("In fn1: {}", a);
    a
    }
```

特殊生存周期

```
fn longest(x: &str, y: &str) -> &str {
      if x.len() > y.len() {
            x
      } else {
            y
      }
}
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but the
signature does not say whether it is borrowed from `x` or `y`
```

特殊生存周期

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

两个智能指针

- std::boxed::Box (有 owership)
- std::rc::Rc (Reference Counted)

Box

```
use std::boxed::Box;

fn main() {
    let a = Box::new(5);
    let b = a;
    let c = a;
}
error[E0382]: use of moved value: `a`
 --> src/main.rs:6:13
  |
4 |     let a = Box::new(5);
  |         - move occurs because `a` has type `std::boxed::Box<i32>`, which does not implement the `Copy` trait
5 |     let b = a;
  |             - value moved here
6 |     let c = a;
  |             ^ value used here after move
```

Rc

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

fn main() {
    let a = Rc::new("abc".to_owned());
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Rc::clone(&a);
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Rc::clone(&a);
        println!("{}", c);
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

小结

- 据说 Rust 是面向 Ownership 编程（需要转变观念）
- 不同于使用 GC 或者自行 (de)allocate， Rust 使用 Ownership 来管理堆上的数据
- 对一份数据， rust 允许有一个可变引用，或者是任意多个不可变引用
- 为了管理比较复杂的引用， rust 里多出了生存周期这一概念
- 智能指针基本上就是在试图提供多种 Owernship 的管理方式
- Ownership 可以避免很多程序中的 bug

Rust 对空指针的零容忍

- std::option::Option
- None
- Some(T)

```
pub struct ListNode {
      pub val: i32,
      pub next: Option<Box<ListNode>>
}
```

小挑战： leetcode 2. Add Two Numbers

```
fn add_two_numbers(l1: Option<Box<ListNode>>, l2: Option<Box<ListNode>>) ->
Option<Box<ListNode>>
```

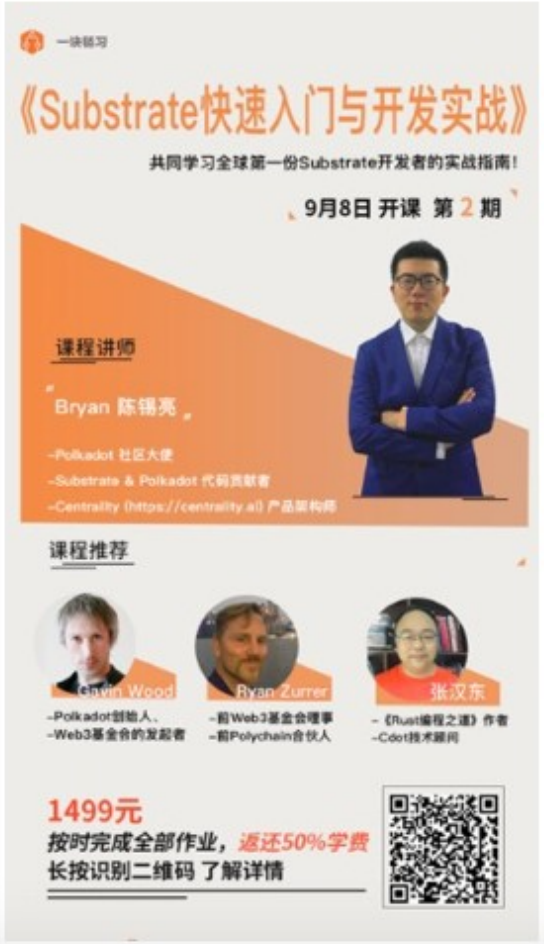
- **2. Add Two Numbers**
- You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order** and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.
- You may assume the two numbers do not contain any leading zero, except the number 0 itself.
- **Example:**
- **Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4) **Output:** 7 -> 0 -> 8 **Explanation:** 342 + 465 = 807.

全球第一份Substrate开发者的实战指南

一块链习

2019年9月上线《Substrate 快速入门与开发实战》训练营课程-1499元

全球第一份 Substrate开发者的实战指南，Polkadot 社区大使亲自授课，并且获得 Polkadot 创始人 Gavin Wood 亲自推荐，培育全球第一批开发者共计 50 余名。第二期即将于 9 月 8 日开课。



课程大纲

课前导读

0.1为什么学习 Substrate
0.2讲师简介
0.3课程介绍
0.4课前准备

第一课（9月8日）

第一课 Substrate架构介绍

1.1.Substrate 基本介绍
1.2.Polkadot 波卡生态圈
1.3.下载编译 Substrate
1.4.启动建立测试网，浏览区块信息，发交易

第二课（9月12日）

第二课 Substrate模块介绍

2.1.Substrate 架构一览
2.2. 常用命令行参数
2.3.Substrate 架构一览
2.4.Substrate Module 模块组成
2.5.项目文件结构简介
2.6.Substrate Kitties 成品展示

第三课（9月19日）

第三课 加密猫简单实现

3.1.创建多节点网络
3.2.macro 宏使用
3.2.1.cargo expand
3.2.2.decl_module
3.2.3.decl_storage
3.3.Substrate Kitties 简单实现

第四课（9月22日）

第四课 JS SDK

4.1.Substrate Kitties V2 实现
4.2.polkadot.js 介绍
4.3.polkadot.js 项目创建

第五课（9月26日）

第五课 数据结构实现

5.1.链表数据结构实现
5.2.单元测试
5.2.polkadot.js 添加自定义类型
5.3.Kitties UI 实现

第六课（9月29日）

第六课 模块间交互

6.1.使用 balances 模块
6.2.Dependency Injection 依赖注入
6.3.Events 事件

第七课（10月6日）

第七课 Substrate底层介绍

7.1.Kitties UI 完全实现
7.2.Metadata 元数据详细介绍
7.3.SCALE 编码

第八课（10月10日）

第八课 上线前的准备和注意事项

8.1.随机数
8.1.1交易费设计
8.1.2代币经济学
8.1.3自治
8.1.4兼容性

扫码报名