

Substrate 区块链应用开发

Rust 枚举和模式匹配

孙凯超
kaichao@parity.io

获取帮助: <https://substrate.io>

内容

● 枚举数据类型简介

- Option

- Result

● 模式匹配的使用

枚举

枚举也是一种**数据类型**，可以用来表示多个**变体**（同一类型的多种可能性），例如

- 交通信号灯，可以红色、绿色或者黄色，但是不可能同时显示两种颜色
- IP 地址使用的可能是 IPv4 协议，也可能是 IPv6 协议

枚举的定义

```
enum TrafficLight {  
    Red,  
    Green,  
    Yellow,  
}
```

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

```
let yellow = TrafficLight::Yellow; let home = IpAddr::V4(String::from("127.0.0.1"));
```

- 变体可以包任意类型的数据，如字符串、数值、结构体、其它的枚举等
- 也可以对枚举定义方法，实现 trait

为枚举定义方法

Demo

枚举：Option<T>

Option 是最常见的一种异常处理机制，它表示两种可能的场景：

- 存在某种类型的值
- 不存在有效的该类型值

Option 定义的方法有 `is_some`, `map`, `map_or`, `unwrap` ...

<https://doc.rust-lang.org/std/option/enum.Option.html>

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

以 `u8` 为例：

```
enum Option_u8 {  
    Some(u8),  
    None,  
}
```

枚举 : Result<T, E>

Result 是另外一种异常处理机制，
它表示两种可能的场景：

- 正确的某种类型的值
- 表示错误信息的另一类型值

Result 定义的方法有 is_ok, map, map_or, unwrap ... }

<https://doc.rust-lang.org/std/result/enum.Result.html>

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

以 u8, String 为例：
enum Result_u8_string
{
 Ok(u8),
 Err(String),
}

模式匹配

Pattern match 可以方便地实现条件分支管理，
if else 的加强版：

- 适用于几乎所有的数据类型
 - 基本类型 u8, bool ...
 - 复杂类型 struct, enum, tuple ...
- 匹配必须完备
- _ (下划线) 可以匹配所有的值

模式匹配举例

```
let value = 0u8;  
match value {  
    1 => println!("one"),  
    3 => println!("three"),  
    _ => (),  
}
```

```
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);
```

特例 if let

- 只关心一个分支时
- 失去了完备性检查

```
let some_u8_value = Some(3u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

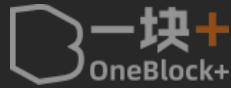
```
if let Some(3) = some_u8_value {
    println!("three");
}
```

Questions?

官网文档 : substrate.io
知乎专栏 : parity.link/zhihu

kaichao@parity.io

Twitter/Wechat: kaichaosun



Substrate 区块链应用开发

Rust 所有权

孙凯超
kaichao@parity.io

获取帮助: <https://substrate.io>

内容

- 所有权概念和规则
- 所有权转移
- Copy & Clone
- 函数与所有权
- Reference 和 borrowing
- Slice 类型

所有权的概念

任何程序的运行都需要依赖内存，典型的**内存管理**机制有：

- 垃圾收集器，如 Java, GO
- 手动分配和释放，如 C/C++
- 编译时的**所有权系统**（Ownership），只有 Rust

Ownership 是 Rust 区别于其它编程语言最核心的特性，它保证了代码的**内存安全性**，且性能卓越。

所有权的规则

- 任何值都有一个变量与之对应，称为 owner
- 某一时刻，只能有一个 owner
- 当 owner 退出作用域后，值被丢弃

```
{
```

```
...
```

```
let s = String::from("hello")
```

s还未定义

定义s并分配存储

```
...
```

```
}
```

s作用域结束，释放存储

所有权的转移 (Move)

```
{  
    ...  
    let s1 = String::from("hello ");  
    let s2 = s1; // 定义s1并分配存储  
    ...  
}  
// s1对应值的所有权转移至s2,s1失效  
// s2作用域结束，释放存储
```

如果是简单类型比如数值, bool, 赋值会发生数据拷贝, 而不是转移所有权。

Copy & Clone

赋值时可以通过**数据拷贝/克隆**，不去转移现有数据所有权，

- **Copy**，适用于基本类型或完全由基本类型组成的复杂类型，
 - 如 u32, bool, char, tuples
- **Clone**，数据存储在堆上，在堆上克隆一份新的，
 - 如 String, HashMap, Vec

Demo

Copy & Clone

```
{  
    ...  
    let s1 = String::from("hello ", 定义s1并分配存储  
    let s2 = s1.clone(); 克隆s1至s2,s1仍然有效  
    ...  
}  
                                         s1,s2作用域结束，分别释放存储
```

- 自动派生Copy或Clone 接口， #[**drvive**(Copy, Clone)]
- 一般不需要显示实现Copy
- Clone 更慢， clone() 不可缺省

函数与所有权

- 和赋值类似，将值传递给函数也会转移所有权或copy
- 返回值可以把函数内变量对应值的所有权转移至函数外

函数与所有权：参数

```
fn main() {  
    let s = String::from("hello");    // 定义s并分配存储  
    takes_ownership(s);            // s对应值的所有权转移  
  
    let x = 5;  
    makes_copy(x);  
}  
  
fn takes_ownership(some_string: String) {    // some_string获得所有权  
    println!("{}!", some_string);  
}  
                                         // some_string作用域结束，释放存储
```

函数与所有权：参数

```
fn main() {  
    let s = String::from("hello");  
    takes_ownership(s);  
  
    let x = 5;  
    makes_copy(x);  
}  
  
fn makes_copy(some_integer: i32) {  
    println!("{}", some_integer);  
}
```

copy x对应的值并传递

some_integer获取copy后的数据

some_integer作用域结束，弹出stack

函数与所有权：返回值

```
fn main() {  
    let s1 = gives_ownership();
```

返回值的所有权从函数转移至s1

```
let s2 = String::from("hello");  
let s3 = takes_and_gives_back(s2);  
}
```

```
fn gives_ownership() -> String {  
    let some_string = String::from("hello");  
    some_string  
}
```

定义并分配存储

返回some_string并转移所有权至函数外

函数与所有权：返回值

```
fn main() {  
    let s1 = gives_ownership();  
  
    let s2 = String::from("hello");  
    let s3 = takes_and_gives_back(s2);  
}  
  
fn takes_and_gives_back(a_string: String) -> String  
{  
    a_string  
}
```

定义并初始化存储

s2对应值所有权转移至函数
s3获得返回值的所有权

a_string获得所有权

返回a_string， 所有权转移至函数外

Reference 和 Borrowing

当需要使用某个值，但又不希望获取所有权时，可以通过引用，

- 在变量名前放置&符号，获取值的引用
- Borrowing：函数参数为引用
- 默认是不可变的（immutable），可变引用为 &mut
- 引用的作用域，在最后使用的地方结束，而不是大括号的末尾。

Reference 和 Borrowing

```
fn main() {  
    let s1 = String::from("hello");      // 定义并分配存储  
  
    let len = calculate_length(&s1);     // borrow变量对应的值  
  
    println!("The length of '{}' is {}.", s1, len);  
}  
  
fn calculate_length(s: &String) -> usize {  
    s.len()                            // 参数s为引用类型  
}
```

s退出作用域，不清空引用的值

Reference 和 Borrowing：可变引用

```
fn main() {  
    let mut s = String::from("hello"), 定义可变变量并分配存储
```

```
        change(&mut s); 创建可变引用并传递给函数  
}
```

```
fn change(some_string: &mut String) { 参数some_string为可变引用  
    some_string.push_str(", world"); 修改值  
}
```

some_string退出作用域，不清空引用的值

Slice 类型

和引用类似， slice 也不拥有值的所有权，用于引用集合内的部分连续数据，

- 与值绑定，当退出作用域，需要清空时，slice 也同时失效
- 定义slice : &name[start..end]，不包含end
- 类型签名 : &str 为 string slice, &[T] 为 Vector / array slice

定义 slice

```
let s = String::from("hello world");
```

定义并分配存储

```
let hello = &s[0..5];  
let world = &s[6..11];
```

定义引用位置[0,5)的str slice

```
let a = [1, 2, 3, 4, 5];  
let slice = &a[1..3];
```

slice引用的内容为 [2,3]

Slice 举例

Demo

内存安全性

所有权和引用如何保证内存安全性？

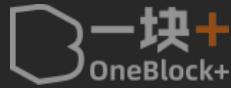
- 拥有所有权的变量退出作用域时，自动清空值的内存空间
- 同一时间，最多有一个可变引用，或者多个不可变引用
- 编译时不允许空指针
- 通过 slice 引用值的一部分

Questions?

官网文档 : substrate.io
知乎专栏 : parity.link/zhihu

kaichao@parity.io

Twitter/Wechat: kaichaosun



Substrate 区块链应用开发

Rust 泛型, trait 和生命周期

孙凯超
kaichao@parity.io

获取帮助: <https://substrate.io>

内容

- 泛型的介绍
- trait 使用
- 生命周期

为什么使用泛型

- 减少相似代码
- 通过抽象，增加扩展性
- 常用于结构体，枚举和函数签名

泛型 - 结构体

- 减少相似代码
- 通过抽象，增加扩展性
- 常用于结构体，枚举和函数签名

```
struct Point<T> {  
    x: T,  
    y: T,  
}  
  
impl<T> Point<T> {  
    fn x(&self) -> &T {  
        &self.x  
    }  
}  
  
fn main() {  
    let integer = Point { x: 5, y: 10 };  
    let float = Point { x: 1.0, y: 4.0 };  
}
```

泛型 - 枚举

- 减少相似代码
- 通过抽象，增加扩展性
- 常用于结构体，枚举和函数签名

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

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

泛型 - 函数签名

- 减少相似代码
- 通过抽象，增加扩展性
- 常用于结构体，枚举和函数签名

```
fn largest<T>(list: &[T]) -> &T {  
    let mut largest = &list[0];  
  
    for item in list {  
        if item > largest {  
            largest = item;  
        }  
    }  
    largest  
}
```

Demo

使用泛型的注意点

- 编译时使用具体类型替代，**不影响执行效率**
- 过多的泛型，**可读性降低**

Traits

- trait 抽象了某种功能或者行为

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

```
pub trait Summary {  
    fn summarize(&self) -> String {  
        format!("read more ...")  
    }  
}
```

Traits

- 对泛型添加 trait bound 表示泛型参数满足某种约束

Demo

```
struct Tweet {  
    author: String,  
    text: String,  
}
```

```
impl Summary for Tweet {  
    fn summarize(&self) -> String {  
        format!("{}: {}",  
               self.author, self.text)  
    }  
}
```

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

// 简单情形时

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

Traits

- 多个参数时， trait bound 保证类型一致

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {  
    println!("{} {}", item1.summarize(), item2.summarize());  
}
```

Traits

- 多个类型约束时，使用 +

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

Traits

- where 关键字

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

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

变量的生命周期

- 每个变量都有生命周期 (**lifetime**)
- 生命周期确保引用的有效性，防止出现空指针

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

引用的生命周期

- 多数情况，可由编译器推断出来
- 推断不出时，使用泛型指定多个引用之间生命周期的关系

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

引用的生命周期

- 返回值的引用生命周期必须来自参数
- 如果来自函数内，会造成空指针，编译不通过

```
fn longest<'a>(x: &str, y: &str) -> &'a str {  
    let result = String::from("really long string");  
    result.as_str()  
}
```

引用的生命周期：缺省规则

- 为每一个引用参数类型，添加生命周期泛型
 - `fn foo<'a, 'b>(x: &'a i32, y: &'b i32);`
- 生命周期泛型只有一个时，所有引用类型的返回值使用此生命周期
 - `fn foo<'a>(x: &'a i32) -> &'a i32`
- 生命周期泛型有多个时，且其中一个为 `&self` 或者 `&mut self`，所有引用类型的返回值使用它对应的生命周期
 - `fn foo<'a, 'b>(&'a self, x: &'b i32) -> &'a i32`

结构体的生命周期

- 此类结构体的作用域不能在引用的值之外

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

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

静态生命周期 (static)

- 此类引用的有效性是程序的整个执行周期
- string literal 默认 static
 - `let s: &'static str = "I have a static lifetime.;"`
- 谨慎使用static，修复代码可能存在的空指针或者引用生命周期不匹配

Questions?

官网文档 : substrate.io
知乎专栏 : parity.link/zhihu

kaichao@parity.io

Twitter/Wechat: kaichaosun



Substrate 区块链应用开发

Rust 项目管理

孙凯超
kaichao@parity.io

获取帮助: <https://substrate.io>

内容

- package, crate, module 介绍
- 功能模块引入
- Workspace

为什么要做项目管理

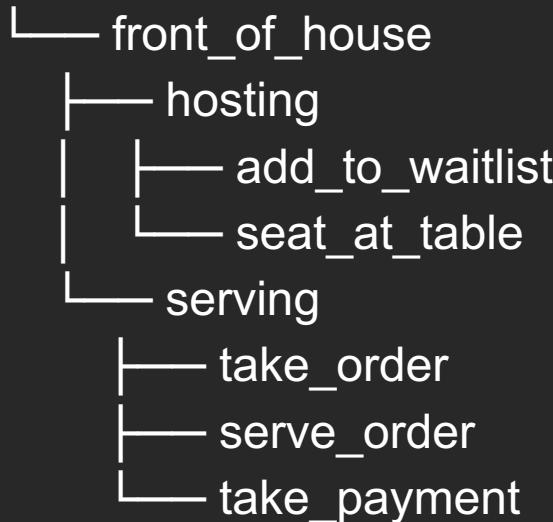
- 组织大量代码
- 封装无需外部关心的实现细节
- 代码复用

Rust 项目管理的组件

- package : cargo 工具用来构建、编译、测试的空间
- crate : 工具库 (src/lib.rs) 或可执行程序 (src/main.rs)
 - rand
 - serde
 - diesel
- module : 在crate里组织代码，控制是否对其它模块可见

Module Tree

crate



模块引入

- use
- pub use
- crate
- self
- super
- as

Demo

Workspace

- 管理多个 library, binary
- 共享cargo.lock和输出目录
- 依赖隔离

Demo

<https://doc.rust-lang.org/book/ch14-03-cargo-workspaces.html>

作业

- 为枚举交通信号灯实现一个 trait， trait里包含一个返回时间的方法，不同的灯持续的时间不同；
- 实现一个函数，为u32类型的整数集合求和，参数类型为 &[u32]，返回类型为Option<u32>，溢出时返回None；
- 实现一个打印图形面积的函数，它接收一个可以计算面积的类型作为参数，比如圆形，三角形，正方形，需要用到泛型和泛型约束。

Questions?

官网文档 : substrate.io
知乎专栏 : parity.link/zhihu

kaichao@parity.io

Twitter/Wechat: kaichaosun