Rust介绍（下）

结构体、枚举与模式匹配

Rust通过`struct`关键字定义一个结构体类型。

在定义完结构体后，可以直接通过字面方式构造结构体的实例。结构体中的成员通过`.`进行访问。

```
struct Foo {
    a: String,
    b: u64,
}

let mut foo = Foo {
    a: String::from("123"),
    b: 123,
};

foo.b = 456;
```

对于字面构造，Rust也提供了一些语法糖：

```
let a = String::from("123");
let foo1 = Foo {
    a, //可以直接使用上下文中的同名变量进行初始化
    b: 123,
};
let foo2 = Foo {
    b: 123
    ..foo1 //这表示剩余没有初始化的成员都使用foo1中的成员进行初始化
};
```

在上面这个例子中，如果我们尝试再使用`foo1`，会发现我们可以正常读取`foo1.b`，但是`foo1.a`却会提示其所有权已经被移动。这是因为Rust中结构体可以**部分移动**，即**移动某个字段**，此时这个结构体**不可以再作为一个整体使用**，但是其他字段还可以正常使用。

另外还有两种结构体定义方式：

```
struct Color(i32, i32, i32); //元组结构体
struct AlwaysEqual; //类单元结构体

let red = Color(255, 0, 0); //字面构造
let red_r = red.0; //和元组一样，通过.0, .1等方式来访问元素

let subject = AlwaysEqual; //由于没有数据，AlwaysEqual的所有实例都是相等的
```

元组结构体可以帮助我们快速定义结构体而不需要成员命名，而类单元结构体则更多强调其绑定的方法，而非类中的数据。

Rust通过`impl`关键字定义类的方法，在定义时，`Self`指代当前要实现的类型，`self`指代类实例本身（不需要指定类型）。对self加上`&`或`&mut`限定可以使用对应的引用，而直接传入`self`则意味着这个函数会消耗实例的所有权。

```rust
impl Foo {
    fn new(a: String, b: u32) -> Self { //这里写-> Foo也一样
        Self {a, b}
    }
}

impl Foo { //可以使用多次impl块
    fn print_members(&self) { //取self的不可变引用
        println!("a: {}, b: {}", self.a, self.b);
    }

    fn into_a(self) -> String { //消耗self的所有权
        self.a
    }
}
```

这里的`new`函数是一个Foo的**关联函数**，因为它没有`self`参数，调用的时候使用`Foo::new`进行调用。

```rust
let foo = Foo::new(String::from("123"), 123); //使用 类::方法 调用关联函数
foo.print_members(); //使用 . 调用类的方法
```

Rust通过`enum`关键字来定义一个枚举类型。枚举既可以是简单的枚举，也可以是带数据的枚举。枚举项通过`::`来访问。

```rust
enum Dimension {
    X, Y, Z //简单枚举，每一项都没有关联数据
}

enum Message {
    Quit, //简单枚举
    Move { x: i32, y: i32 }, //带有数据的枚举项，和一般结构体类似
    Write(String), //带有数据的枚举项，和元组结构体类似
    ChangeColor(i32, i32, i32), //带有数据的枚举项，和元组结构体类似
}

impl Message {
    fn handle(&self) {} //一样可以通过impl来定义枚举类的方法
}

let quit = Message::Quit; //使用::获得枚举项
let move = Message::Move{x: 1, y: 0};
let write = Message::Write(String::from("123"));
let changeColor = Message::ChangeColor(255, 0, 0);
```

> 这种带数据的枚举项提供了比C/C++枚举更强的表达能力。

这里介绍几个特殊的枚举：`Option`以及`Result`，Rust会预导入它们供用户直接使用【不需要使用`::`，直接使用枚举项类型】。

在实际开发中经常会遇到**函数并不能确定是否一定有数据返回**的情况，此时就可以用Option进行包裹，以满足返回值类型要一致的约束。而`Result`更多用于错误处理。

```rust
enum Option<T> { //T是泛型参数
    None,
    Some(T),
}
```

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

fn safe_divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None //除数为0，返回None
    }
    else {
        Some(a / b) //正常返回结果
    }
}
```

对于函数的返回结果，我们要怎么判断到底是None还是有数据的Some呢？Rust提供了强大的模式匹配语法。可以使用`match`表达式对枚举类的实例进行匹配判断。

> 可以看作是C/C++中switch的加强版。

```
let result = safe_divide(a, b);

match result {
    None => println!("被除数为0! "),
    Some(val) => println!("结果为: {}", val),
};

let message_hash = match message {
    Message::Quit => -1, //每个匹配项以,结尾
    Message::Move{x, y: _y} => x * 10 + _y, //x直接匹配成员x，成员y对应的值则指定由_y表示
    Message::Write(s) => { //一样可以用大括号表达式，s是String类型
        println!("{}", &s);
        -(s.len() as i32)
    },
    Message::ChangeColor(r, ..) => r, //..代表后面的匹配项省略，也可以写成(r, _, _)
}
```

在使用match表达式对枚举进行匹配的时候，Rust编译器强制要求**match中的分支要覆盖所有的可能性**，这一检查极大的提高了代码的可维护性。

除了对枚举实例进行匹配，match还可以对普通的数值进行匹配。

```
let a: i32 = 9;
match a {
    1 => println!("1"),
    other => println!("{}", other * 2), //other作为变量名匹配a本身的值
};

match a {
    1 => println!("1"),
    _ => println!("Not 1"), //_类似default，表示前面分支均不满足的情况
}
```

如果我们只关注某一个分支，使用match似乎有点小题大作。此时可以使用`if let`来进行某一情况的模式匹配。

```rust
let v = Some(1);
if let Some(val) = v { //一样的模式匹配，左边是模式，匹配成功后进入分支
    println!("{}", val);
}
// 一样可以用else
```

练习1：写一个交通灯`TrafficLight`类，其包含两个成员`color`和`time_remain`。分别表示当前灯的颜色和剩余时间。要求实现`next`函数，每次调用`next`，剩余时间减去1。当时间为0时，变更颜色并重新置数（具体数值不限）。

参考代码：

```rust
enum Color {
    Red, Green, Yellow
}
struct TrafficLight {
    color: Color,
    time_remain: u32,
}
impl TrafficLight {
    fn next(&mut self) {
        if self.time_remain <= 1 {
            (self.color, self.time_remain) = match self.color {
                Color::Red => (Color::Green, 60),
                Color::Green => (Color::Yellow, 5),
                Color::Yellow => (Color::Red, 55),
            };
        }
        else {
            self.time_remain -= 1;
        }
    }
}
```

泛型、trait与生命周期

Rust的泛型系统也很强大，普通函数、类以及类的方法都可以使用泛型来复用代码。我们先来看泛型函数如何定义。

```rust
fn largest<T>(list: &Vec<T>) -> &T {
```

可以看到，函数需要使用的泛型类型被放在**函数名后的尖括号内**，而参数表和返回值类型则都可以用泛型类型。

如果我们用下面的代码试着实现这个函数，会发现编译器报错：

```rust
fn largest<T>(list: &Vec<T>) -> &T {
    let mut v = &list[0];
    for item in list {
        if item > v { // Error!  binary operation `>` cannot be applied to type
`&T`
            v = item;
        }
    }
    v
}
```

这是因为我们对泛型T没有做任何的约束，不可能每个传入的类型都可以使用`>`相互比较。在报错信息中，编译器建议使用`std::cmp::PartialOrd`来对泛型做一个限制，这里的`PartialOrd`就是一个用于实现比较关系的trait。

在讲trait之前，先看看类和其方法的泛型如何定义。之前的`Option<T>`就是一个拥有泛型参数的枚举类，而`Result<T, E>`则是拥有两个泛型参数的枚举类。

```rust
struct Point<X, Y> { //两个泛型参数
    x: X,
    y: Y,
}

impl Point<f32, f32> { //为具体类型实现方法
    fn distance_from_origin(&self) -> f32 { //计算到原点的距离
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}

impl<X1, Y1> Point<X1, Y1> { //在impl之后声明泛型参数，可以为【泛型类型】实现方法
    fn x(&self) -> &X1 { //方法可以直接使用impl所声明的泛型
        &self.x
    }

    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> { //方法也可以引
入新的泛型参数
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

let p1 = Point {x: 5, y: 10.4}; //Point<i32, f32>
let p2 = Point {x: String::from("3"), y: 'c'}; //Point<String, char>
let p3 = p1.mixup(p2); //Point<i32, char>
println!("{} {}", p3.x, p3.y);
```

练习2：利用`Vec<T>`实现泛型栈`Stack<T>`，具有`push`和`top`方法，在`top`方法中需要检查是否栈空。

参考代码：

```rust
struct Stack<T> {
    data: Vec<T>
```

```
}

impl<T> Stack<T> {
    fn push(&mut self, value: T) {
        self.data.push(value);
    }

    fn top(&self) -> Option<&T> {
        let len = self.data.len();
        match len {
            0 => None,
            _ => Some(self.data[len - 1]),
        }
    }
}
```

Rust中的trait和其他语言的interface很像（例如Java），用来定义一组共用的行为。利用trait限制泛型参数，我们可以写出更加严格的泛型代码。

下面是一个`trait`定义的例子。

```
trait ToString {
    const CONSTANT: i32; //关联常量，实现trait时必须赋值
    type Item; //关联类型，实现trait时必须指定

    fn to_string(&self) -> String; //只有方法声明，实现trait时需要手动实现

    fn print(&self) { //方法有默认实现，可以不手动实现
        println!("{}", self.to_string());
    }
}
```

如果我们要为某个类型实现该trait，一样使用`impl`关键字。

```
struct Foo<T> {
    tag: String,
    v: T,
}

impl<T> ToString for Foo<T> {
    const CONSTANT: i32 = 42; //仅作演示，没有实际意义
    type Item = i32; //仅作演示，没有实际意义

    fn to_string(&self) -> String {
        self.tag.clone()
    }
}
```

之前提到过的泛型约束，需要在泛型参数表中使用`:`明确指定：

```
fn notify<T: ToString>(item: &T) { //指定类型T必须要实现ToString特征
    item.print();
}
```

当然还有一种语法糖的形式：

```
fn notify(item: &impl ToString) { //指定参数类型必须要实现ToString特征
    item.print();
}
```

这里`impl ...`作为一个类型会被编译器在编译时替换成泛型类型，它也可以用在返回值类型中。

当我们需要泛型类型满足多个trait约束时，可以使用`+`列举这些trait约束。核心库中的`Display` trait可以使对象能够被`println!`以及`format!`这样的格式化字符串使用。这里举个例子：

```
use core::{cmp::PartialOrd, fmt::Display}; //引入这两个trait的符号

fn largest<T: Display + PartialOrd>(list: &Vec<T>) -> &T {
    let mut v = &list[0];
    for item in list {
        if item > v { //因为T实现了PartialOrd，因此可以做偏序比价
            v = item;
        }
    }
    println!("{}", v); //因为T实现了Display，因此可以用在格式化输出中
    v
}
```

当trait约束过多时，函数声明部分会变得非常长，因此Rust也提供另一种先声明泛型参数，最后用`where`指出约束的形式：

```
fn largest<T>(list: &Vec<T>) -> &T
where
    T: Display + PartialOrd
{
    let mut v = &list[0];
    for item in list {
        if item > v {
            v = item;
        }
    }
    println!("{}", v);
    v
}
```

trait当然也是可以有泛型的，使用方式和其他泛型区别不大，理解了impl的写法这种泛型也不难实现，这里不再做赘述。

下面介绍Rust借用的**生命周期**机制。Rust中每个借用都具有其生命周期，大多数情况下，借用的生命周期可以由编译器自己进行推导。

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

这里`'a`标注的是`r`的借用的生命周期，而`'b`标注的是`x`的生命周期。可以看到在`r = &x`处代码尝试让一个短生命周期的引用生命周期扩大到`'a`（赋值给生命周期为`'a`的`r`），这在语义上当然是不允许的，因此编译器会直接报错。

而如果我们将代码改成下面的样子，就不会有问题，因为赋值操作在**缩短生命周期**而非扩大，这是正确的语义。

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

在函数体中，这种分析非常简单，而当引用作为跨函数传递的数据时，事情开始变得复杂起来（每个函数是独立编译的，但是分析生命周期却需要跨函数分析）。

```
fn longest(x: &str, y: &str) -> &str { //Error! 编译器无法判断返回值到底是x还是y
    if x.len() > y.len() { x } else { y }
}


//下面的代码说明为什么要在这些引用中做生命周期的区分
let result;
let x = String::from("hello");  // x 的有效作用域

{
    let y = String::from("world!!!!");  // y 的作用域更短
    result = longest(&x, &y);  // result 可能借用 x 或 y
    println!("在内部作用域: {}", result);
    // y 在这里被释放
}

// 如果 result 指向 y，这里就会出问题
println!("在外部作用域: {}", result);  // 潜在悬垂引用
```

> 这里str是String的不可变切片类型，&str就是一段不可变字符串切片的借用

由于`longest`的返回值既可能是x也可能是y，那么编译器在调用函数处就没有办法直接检查是否合法。在上面的代码示例中，第一次print，result是合法的，而第二次print就不合法了，因为y已经被释放，result成了**悬垂引用**。

使用生命周期泛型，我们可以给函数添加额外的生命周期**约束**用于编译器的检查。

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

这里的意思是说，我们定义了一个生命周期泛型`'a`，x和y都需要被这个`'a`生命周期约束（即x和y的生命周期都>=`'a`）。假设它们的生命周期分别为L_x和L_y，那么这里$'a \subset L_x \cap L_y$，因为要求x和y都能缩短到`'a`。而返回出来的引用被使用的范围R则需要在`'a`的范围内，即$R \subset' a \subset L_x \cap L_y$。

> 用不严谨的话说：
>
> - 赋值/传参时，借用的生命周期只允许缩小，不允许放大。
> - 条件1：x和y从外部赋值到函数内部 => x,y生命周期 >= `'a`
> - 条件2：拥有`'a`生命周期的返回值返回出去，在外面赋值给了新的引用 => `'a` >= 新引用的生命周期
> - 综合下来就得到结果：返回借用的使用范围应该在参数生命周期的交集中

一个更好的理解方式是：Rust中的借用存在**Reborrow**，编译器会将借用的生命周期缩短直到适配，**生命周期的作用就是标记重借用时不同引用的相对关系，而非代表某个固定的范围。**

```rust
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {result}"); // correct
    }
}
```

上面的代码中显然string1和string2的引用的生命周期都可以缩短到result这个借用的生命周期（'a），因此当然没问题。

再看几个例子：

```rust
fn longest<'a>(x: &'a str, y: &str) -> &'a str { //Correct! 返回的借用在函数体内只与x有关
    x
}

fn longest<'a>(x: &str, y: &str) -> &'a str { //Error! 返回的借用与参数没有关系，把局部变量的悬垂引用返回出去本来就非法
    let result = String::from("really long string");
    result.as_str()
}
```

生命周期在Rust中也是**类型系统**的一部分，泛型参数可以有trait约束，而生命周期作为泛型参数也可以指定约束关系：

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

`'b: 'a`的意思是：`'b`（长的）被`'a`（短的）约束，`'b`可以缩短到`'a`。另外`where`的写法这里也一样可以使用。

使用生命周期泛型，我们也可以把借用塞到结构体中去。

```
struct Parser<'a> {
    input: &'a str,
    context: &'a str,
}
```

理解这里生命周期的方式是一样的：Parser实例的使用范围需要小于input和context生命周期的交集。（这两个借用的生命周期都可以缩小到Parser）

而如果input和context之间没有约束需求，那可以写成

```
struct Parser<'a, 'b> {
    input: &'a str,
    context: &'b str,
}
```

这样当我们在实现Parser的方法时，可以有更精确的生命周期约束，例如单独返回input成员或context成员。

> Rust编译器是如何补充函数中借用的生命周期的?
>
> - 首先为所有借用**参数**加上各不相同的生命周期约束
> - 如果参数只有一个，那么返回值的生命周期约束与参数相同
> - 如果参数存在self，那么返回值的生命周期约束与self相同
>
> 因此形如`fn longest(x: &str, y: &str) -> &str`这样的函数会报错。

最后提一下一个特殊的lifetime：`'static`，它表示借用的生命周期和整个程序一样长。

练习3：写一个`TextChosser`结构体，其保存一个`str`引用，要求实现`new`，`choose`方法，后者和传入的另一个`&str`做比较，返回长度较长的引用（相等则任意一个均可）。

参考代码：

```
struct TextChooser<'a> {
    base: &'a str,
}

impl<'a> TextChooser<'a> {
    fn new(s: &'a str) -> Self {
        TextChooser {base: s}
    }

    fn choose(&self, other: &'a str) -> &'a str {
        if self.base.len() > other.len() {
            self.base
        }
        else {
            other
        }
    }
}

fn main() {
    let s1 = String::from("hello");
```

```
    let chooser = TextChooser::new(&s1);

    {
        let s2 = String::from("world!!");
        let result = chooser.choose(&s2); //为什么s2的生命周期短于s1以及chooser但依旧没问题？'a只是标记，实际发生了重借用，都被缩短到了result的生命周期，显然s1s2都覆盖了result。
        println!("{}", result);
    }
}
```