TinTin

第三课：复合类型

Mike Tang

daogangtang@gmail.com

2023-5-19

结构体

结构体往往是一个程序的骨干。

```
User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
};
```

结构体的更新

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

更新时的便捷写法

```
fn main() {
    let active = true;
    let username = String::from("someusername123");
    let email = String::from("someone@example.com");
    let user1 = User {
        active,
        username,
        email,
        sign_in_count: 1,
    };
}
```

基于已有的实例的值来创建新实例，并只更新部分字段

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=98ac30401e3f494effed158314ab4af2

元组结构体

所谓元组结构体，也就是元组和结构体的结合体！就是下面这个样子

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

元组结构体有类型名，但是无字段名，也即字段是匿名的。这在有些情况下很有用，因为想名字很头痛，并且明显元组结构体更紧凑。上述示例元组部分其实是一样的，但是类型名不同，它们就是不同的类型。

单元结构体

```
struct ArticleModule;


fn main() {
    let module = ArticleModule;
}
```

单元结构体就是只有一个类型名字，没有任何字段的结构体。定义和创建实例时连后面的花括号都可以省略。可以看到，上面示例中在使用let语句作绑定时，类型实际创建了一个结构体的实例。这种写法非常紧凑，要注意分辨，不然会疑惑类型为啥能直接赋给一个变量。

面向对象的特性

Rust不是一门面向对象的语言，但是其确实有部分面向对象的特性。Rust承载面向对象特性的部分一般是结构体。Rust有个关键字 impl 可以用来给结构体（或其它类型）实现方法，也就是关联在某个类型上的函数。

方法（实例方法）

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=ea4fc2585ad6388ed6f0dc9fefcc6540

前面我们详细解析过引用，那么，impl的时候，也会对应三种引用，扩展上面的例子：

```
impl Rectangle {
    fn area1(self, n: u32) -> u32 {
        self.width * self.height * n
    }
    fn area2(&self, n: u32) -> u32 {
        self.width * self.height * n
    }
    fn area3(&mut self, n: u32) -> u32 {
        self.width * self.height * n
    }
}
```

方法是实现在类型上的一类特殊的函数，它的第一个参数为Self类型，包含 self: Self, self: &Self, self: &mut Self 三种情况，因为是标准用法了，所以Rust帮我们简写了。上述代码展开后是：

```
impl Rectangle {
    fn area1(self: Self, n: u32) -> u32 {
        self.width * self.height * n
    }
    fn area2(self: &Self, n: u32) -> u32 {
        self.width * self.height * n
    }
    fn area3(self: &mut Self, n: u32) -> u32 {
        self.width * self.height * n
    }
}
```

分别对应把实例所有权传进去，传实例的不可变引用进去，传实例的可变引用进去，三种情况。

这类方法调用的时候，只需要写成下面这样就行了。第一个参数就是调用者实例，会默认传进去：

```
rect1.area1(n);

rect1.area2(n);

rect1.area3(n);
```

看到这里，是不是感觉很熟悉。C++, Java 等的this指针即视感。不过，在Rust中，一切基本上都是显式化的，不存在隐藏提供一个参数给你的情况，统统要写清楚，不会存在坑。是不是很**清晰**。

实例的引用

实例的引用也是可以直接调用方法的，比如，对于不可变引用，要以像下面这样调用。Rust会自动做正确的多级解引用操作。

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=a0b8232929baa8f1e6dfc8adfc6f95a0

对同一个类型，impl可以分开写多次。这在组织代码的时候，有时会带来方便。

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

关联函数（静态方法）

当然，在类型上实现方法，也可以第一个参数不带self参数。这种方法称为关联方法，比如：

```
impl Rectangle {
    fn numbers(rows: u32, cols: u32) -> u32 {
        rows * cols
    }
}
```

调用的时候，用路径符来调用：

```
Rectangle::numbers(10, 10);
```

是不是跟 C++, Java中的静态方法起类似的作用？但是Rust这里不需要额外的static修饰符了。

构造函数

Foo::new();

Foo::from();

Default

TinTin

Newtype模式

```
struct MyVec(Vec<u8>)
```

TinTin

enum

TinTin

枚举这种类型容纳选项的可能性，每一种可能的选项都是一个变体。

Rust中的枚举非常强大，可能是最强大的数据结构了。enum就像一个筐，什么都往里面装。

最强大的复合类型

TinTin

enum中的变体（variant）可以作为名字附带各种形式的结构。什么元组结构体，结构体，也可以作为enum的一个变体存在。


```
enum WebEvent {
    // An `enum` variant may either be `unit-like`,
    PageLoad,
    PageUnload,
    // like tuple structs,
    KeyPress(char),
    Paste(String),
    // or c-like structures.
    Click { x: i64, y: i64 },
}
```

实例化变体

TinTin

实例化变体的时候，也是一致的写法：

```
let a = WebEvent::PageLoad;

let b = WebEvent::PageUnload;

let c = WebEvent::KeyPress('c');

let d = WebEvent::Paste(String::from("batman"));

let e = WebEvent::Click { x: 320, y: 240 };
```

类C枚举

也可以定义C那样的枚举。

```
// enum with explicit discriminator
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}
```

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0107c6db2b651b75aaca551d9d684fcf

空枚举

也可以定义空枚举：

enum MyEnum {};

其实与空结构体一样，都表示一个类型。

但是它不能被实例化：

enum Foo {}

let a = Foo; // 错误的expected struct, variant or union type, found enum `Foo`

not a struct, variant or union type

Impl enum

枚举同样能够被 impl。

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=a6c1d36e91f0abb66d9b858fd02e9d50

Impl enum

但是不能对枚举的变体直接 impl

```
enum Foo {
  AAA,
  BBB,
  CCC
}
impl Foo::AAA {   // 错误的
}
```

模式匹配

TinTin

一般情况下，枚举更多的是用来作配置，并结合 match 使用。下面我们就进入模式匹配的环节。

Rust中的模式匹配非常强大。这个概念直接来自于函数式语言Haskell等。意思就是按对象值的结构形态进行匹配。

使用match来做分支流程

初看有点类似于C/C++/Java 的 switch .. case 。但实际很不一样。

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=3263d9d00bdec1e596ab7e7f26e0dace

Match结合枚举

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0c554a8eb1fe26db3c512eb9e3d9edd1

从上面例子可以看到，match 是可以返回值的。

全部分支必须处理

如上示例。

_ 占位符

TinTin

有时，确实想测试一些东西，或就是不想处理一些分支，可以用 _ 偷懒。

match实际是模式匹配的入口。

if let

TinTin

只有两个分支或在这个位置先只想处理一个分支的情况，

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

if let

TinTin

就可以使用 if let.

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

并且相对于match，在分支体代码比较多的情况下，if let 可以少一层括号。

while let

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=0d08d066da340d8db46f57db8419aa2c

let

TinTin

let本身就支持模式匹配，实际前面if let, while let都是用的let的能力。

匹配元组

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=28c32300d5ac78914f64b3e3739d6538

匹配元组

这种用法，常常叫作元组体的析构。常用来从函数的多返回值中取出数据。比如：

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=285ed83a9cfb4df34356127923b87d0a

匹配枚举

TinTin

```
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
        }
    }
}
```

匹配结构体

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=2ea883b8da0c18cf266ffcddd144f732

如果我们想再用一下a会是什么效果?

函数中的模式匹配

TinTin

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=39746546513d79053e3f48e2f681a6ae

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=c98b78fc58d53ab72b17ce16d090dcea

Vec与HashMap的一些经验

TinTin

- 切片
- 元素所有权

引用到切片引用的自动转换

&String -> &str

&Vec<u8> -> &[u8]

&Vec<T> -> &[T]

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=c2430367e27d846ca26a4a1117a19556

Vec中的所有权

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=08349953f10e74e08727339e0834cf2b

Vec<String> 是对String带有所有权的。那Vec<>中，自然也能放 Vec<&str> 这种引用。

可以看到，所有权不能move出来，只能使用引用去访问。

HashMap中的所有权

HashMap是不会把里面内容的所有权让出来的。

Insert, entry, get, get_mut, iter, iter_mut

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=70356a11c153c3cc4163944a6da515ec

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=c0f187bbd9e6ff907a8be784ac94e97e

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=ae01217bfc7fc59a95715e087ac4373c

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=24b00e7021bca832f57b2b7e051c3627

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=57118410cf09bab756f4aaee1e72f18a

https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=f386499fdd50a2500dc37fb1d852f3dd

Vec和HashMap中的元素所有权如何拿出来

通过迭代器 .into_iter()

Q&A

TinTin