

08 枚举和模式匹配

枚举

在 Rust 中，枚举（Enumeration）类型是很常见的。定义一个枚举类型非常简单：

```
1 enum TimeUnit {
2     Seconds,
3     Minutes,
4     Hours,
5     Days,
6     Months,
7     Years,
8 }
```

如上图所示的，枚举类型 `TimeUnit` 有 6 个不同的值。默认情况下，在内存中映射到从 `0` 开始的整数值。但是我们也可以给枚举值指定不同值：

```
1 enum HttpStatus {
2     Ok = 200,
3     NotModified = 304,
4     NotFound = 404,
5     //...
6 }
```

在其他常用的语言中，也支持枚举类型，比如 Java 也有枚举类型。但是 Rust 中的枚举类型比 Java 的枚举类型更加强大，甚至可以说，Java 中的枚举类型无法和 Rust 的类比。因为，Rust 中的枚举是可以携带数据的，并且可以携带不同类型的数据。这就给 Rust 的枚举类型带来了极大的能力，而且代码也更加清晰。

假设，我在使用 Rust 编写一个机器人对战的程序，机器人会收到消息，然后根据消息采取下一步动作。可能收到的消息有：退出、移动、向屏幕输出消息、改变自己的涂装颜色。

如果用 JavaScript 写，那么可能的代码如下：

```
1 // JavaScript 代码
2 const MessageType = Object.freeze({
3     Quit: 0,
4     Move: 1,
5     Write: 2,
6     ChangeColor: 3,
7 });
8
9 class Message {
10     #type;
11     #data;
12     constructor(type, data) {
13         this.#type = type;
14         this.#data = data;
15     }
16
17     get type() {
18         return this.#type;
19     }
20
21     get data() {
22         return this.#data;
23     }
24 }
25
26 class Robot {
27     onMessage(msg) {
28         if (msg.type === MessageType.Move) {
29             const { x, y } = msg.data;
30             //...
31         }
32
33         if (msg.type === MessageType.ChangeColor) {
```

```

34         const { r, g, b } = msg.data;
35         // ...
36     }
37
38     if (msg.type === MessageType.Write) {
39         const { content } = msg.data;
40         // ...
41     }
42
43     if (msg.type === MessageType.Quit) {
44         // ...
45     }
46 }
47 }
```

虽然代码也还算简洁，但是针对不同的消息，要读取不同的数据。消息的发送端和接收端要协商好消息携带的数据。

在 Rust 中，可以这样写：

```

1 enum Message {
2     Quit,
3     Move {x: i32, y: i32},
4     Write(String),
5     ChangeColor(i32, i32, i32),
6 }
7
8 struct Robot {
9     name: String,
10 }
11
12 impl Robot {
13     fn on_message(&mut self, msg: Message) {
14         match msg {
15             Message::Move { x, y } => println!("I'm going to
move: ({}, {})", x, y),
16             Message::Write(s) => println!("Hello {}", s),
17         }
18     }
19 }
```

```

17     Message::ChangeColor(r, g, b) => println!("I'm
18     changing my color to RGB({}, {}, {})", r, g, b),
19
20   }
21 }
22
23 fn main() {
24   let msges = [
25     Message::Write("Hello you!".to_string()),
26     Message::Move { x: 1, y: 2 },
27     Message::ChangeColor(255, 255, 128),
28     Message::Quit
29   ];
30
31   let mut robot = Robot {
32     name: "Droid".to_string()
33   };
34
35   for msg in msges {
36     robot.on_message(msg);
37   }
38 }
```

很强大，直接将消息的数据携带到消息里面。只要收到了某种类型的消息，那么一定会携带对应的数据，不用担心出现不正确的数据了。正如上面的代码所示，枚举经常和模式匹配（`match`）搭配使用，下一节课我们将会讲解到这部分的内容。

Rust 中的枚举类型不仅可以携带不同类型的数据，还可以拥有自己的方法，例如：

```

1 enum HttpStatus {
2   Ok = 200,
3   NotModified = 304,
4   NotFound = 404,
5   //...
6 }
7
```

```

8  impl HttpStatus {
9      fn from_u32(n: u32) -> Option<HttpStatus> {
10         match n {
11             200 => Some(HttpStatus::Ok),
12             304 => Some(HttpStatus::NotModified),
13             404 => Some(HttpStatus::NotFound),
14             _ => None
15         }
16     }
17 }
18
19 fn main() {
20     let status = HttpStatus::from_u32(200);
21 }
```

再来看一个真实的案例，Json 枚举的定义：

```

1 enum Json {
2     Null,
3     Boolean(bool),
4     Number(f64),
5     String(String),
6     Array(Vec<Json>),
7     Object(Box<HashMap<String, Json>>)
8 }
```

总之，Rust 中的枚举类型非常强大，有着极高的出镜率。

回顾

1. 枚举类型可以携带不同类型的值
2. 枚举类型可以拥有自己的方法

