

# 09 模式匹配

---

在 Rust 中的模式匹配（Pattern Matching）是非常强大的，我们在前面的内容中也体验过一些简单的模式匹配了，例如在上一节课中讲解枚举类型的时候，我们就频繁的使用 `match` 表达式。下面我们通过一些示例代码来展示 `match` 的用法。

## 字面量、变量和通配符模式

```

1  match meadow.count_rabbits() {
2      0 => {}, // nothing to say
3      1 => println!("A rabbit is nosing around in the clover."),
4      n => println!("There are {} rabbits hopping about in the
5          meadow", n),
6  }

```

在上面的例子中，如果 `meadow.count_rabbits()` 的返回值是 `0`，那么就什么都不做；如果是 `1`，就打印一段文字；如果是其他的值，将这个值复制（如果是未实现 `Copy` trait 的类型，那么就是移动）到变量 `n` 中，然后我们在最后一个分支中就可以使用这个变量 `n` 的值了。

字面量除了支持数值之外，也支持其他类型的值，例如下面这个例子，使用 `&str` 作为要匹配的值：

```

1  let calendar = match settings.get_string("calendar") {
2      "gregorian" => Calendar::Gregorian,
3      "chinese" => Calendar::Chinese,
4      "ethiopian" => Calendar::Ethiopian,
5      other => return parse_error("calendar", other),
6  };

```

从上面的两个例子中不难猜出，变量 `n` 和 `other` 就像 JavaScript 中 `switch` 语句的 `default` 分支，或者 `if` 语句的 `else` 分支。也就是说，Rust 中的 `match` 表达式必须是穷尽的。那么问题来了，如果可能的值太多，怎么穷尽？此时就需要用到通配符模式，使用 `_` 来接收其他的值，例如：

```

1 let caption = match photo.tagged_pet() {
2     Pet::Tyrannosaurus => "RRRAAAAHHHHHH",
3     Pet::Samoyed => "*dog thoughts*",
4     _ => "I'm cute, love me", // generic caption, works for any
5     pet
6 };

```

通配符 `_` 表示可以匹配任意值，但是不把匹配到的值存储到变量中。

## 元组和结构体模式

如果在一个 `match` 表达式里面要处理多个值，可以使用元组匹配模式：

```

1 use std::cmp::Ordering::*;
2
3 fn describe_point(x: i32, y: i32) -> &'static str {
4     match (x.cmp(&0), y.cmp(&0)) {
5         (Equal, Equal) => "at the origin",
6         (_, Equal) => "on the x axis",
7         (Equal, _) => "on the y axis",
8         (Greater, Greater) => "in the first quadrant",
9         (Less, Greater) => "in the second quadrant",
10        _ => "somewhere else",
11    }
12 }

```

在上面这个例子中，实际上是在检测一个给定的点在坐标系的什么位置。我们一次匹配 `x` 和 `0` 以及 `y` 和 `0` 比较的结果，此时可以使用元组模式。那么比较的结果也是一个元组。`match` 中的各个分支的含义如下：

- 如果 `x` 和 `y` 都等于 `0`，那么表明是在原点
- 如果 `y` 等于 `0`，表明是在 `x` 轴
- 如果 `x` 等于 `0`，表明是在 `y` 轴
- 如果 `x` 和 `y` 都大于 `0`，表明是在第一象限
- 如果 `x` 小于 `0` 且 `y` 大于 `0`，表明是在第二象限
- 其他情况，就不管了

如果是用 JavaScript 采用比较传统的 `if-else-if` 呢？代码可能会是下面这个样子的：

```

1  function describePoint(x, y) {
2      if (x === 0 && y === 0) {
3          console.log("at the origin");
4      } else if (y === 0) {
5          console.log("on the x axis");
6      } else if (x === 0) {
7          console.log("on the y axis");
8      } else if (x > 0 && y > 0) {
9          console.log("in the first quadrant");
10     } else if (x < 0 && y > 0) {
11         console.log("in the second quadrant");
12     } else {
13         console.log("somewhere else");
14     }
15 }
```

很明显，`match` 表达式更加的简洁清晰。当然了，`match` 表达式也是遇到第一个满足条件的分支之后就不再往下进行比对了，所以分支的前后关系和 `if-else-if` 是一样的，要搞清楚判断的顺序。

结构体匹配和元组匹配很相似：

```

1 match balloon.location {
2     Point { x: 0, y: height } =>
3         println!("straight up {} meters", height),
4     Point { x: x, y: y } =>
5         println!("at ({}m, {}m)", x, y),
6 }

```

在这个示例中，第一个分支匹配了 `x` 是 `0` 的坐标，并且把 `y` 的值接入到变量 `height` 中。如果结构体的属性很多，但是我们只关心个别的属性呢？那就使用 `..` 忽略掉不关心的属性：

```

1 match get_student(id) {
2     Some(Student {
3         active: false,
4         ..
5     }) => println!("student is inactive"),
6     _ => println!("student is active")
7 }

```

## 数组和切片模式

数组和切片的主要区别就是数组是定长的，切片是变长的。数组和切片的模式匹配用来匹配某个位置的值。例如，下面一段代码是把 HSL 的颜色转换成 RGB 颜色的，其中，HSL 颜色是一个 `[u8; 3]` 的数组：

```

1 fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {
2     match hsl {
3         [_, _, 0] => [0, 0, 0],
4         [_, _, 255] => [255, 255, 255],
5         //...
6     }
7 }

```

`match` 的第一个分支（行 3）表示，只要 HSL 的 L 是 `0`，那么转换到 RGB 就是 `[0, 0, 0]`，第二个分支（行 4）表示，只要 HSL 的 L 是 `255`，那么转换到 RGB 就是 `[255, 255, 255]`。

由于切片是变长的，所以当使用切片模式的时候，可以使用 `..` 来匹配任意长度的字符。

```

1 fn greet_people(names: &[&str]) {
2     match names {
3         [] => { println!("Hello, nobody.") },
4         [a] => { println!("Hello, {}.", a) },
5         [a, b] => { println!("Hello, {} and {}.", a, b) },
6         [a, .., b] => { println!("Hello, everyone from {} to
7             {}", a, b) }
8     }

```

第一个分支，匹配到长度为 `0` 的切片；第二个分支，匹配到长度为 `1` 的切片，并且将切片唯一的元素值接收到变量 `a` 中；第三个分支，匹配到长度为 `2` 的切片，并且将下标为 `0` 和 `1` 的值分别接收到变量 `a` 和 `b`。第四个分支，匹配长度大于 `2` 的切片，只接收第一个值和最后一个值到变量 `a` 和 `b`。

## 引用模式

我们在前面提到过，在 `match` 表达式中，匹配一个非拷贝的值，会发生所有权转移的情况。例如下面这段代码：

```

1 // 以下代码无法通过编译
2 match account {
3     Account { name, language, .. } => {
4         ui.greet(&name, &language);
5         ui.show_settings(&account); // error: borrow of moved
6         value: `account`
7     }

```

上面的代码中，`account` 的值如果匹配到了第一个分支，那么 `account.name` 和 `account.language` 的所有权被转移到块级变量 `name` 和 `language` 中（从属性名可以猜测，这两个属性是 `String` 类型的，不可拷贝），`account` 中的其他属性将被丢弃。

那么如何解决这个问题呢？Rust 的模式匹配提供了两种方式：`ref` 模式用来引用部分值，`&` 模式用来匹配引用。比如，上面的代码，我们使用 `ref` 模式改写一下：

```

1  match account {
2      Account { ref name, ref language, .. } => {
3          ui.greet(&name, &language);
4          ui.show_settings(&account);
5      }
6  }
```

现在，本地变量 `name` 和 `language` 是一个指向 `account.name` 和 `account.language` 的引用，就不会发生所有权转移的问题了。

接下来，我们再看看 `&` 模式。假定我们有一个对象 `Rectangle`，表示在平面坐标系中的一个矩形，它有一个方法 `center()` 返回矩形的中心点的坐标 `Point(x, y)` 的引用，也就是 `&Point` 类型。我们如何针对引用类型进行模式匹配呢？

```

1  struct Point(f32, f32);
2
3  struct Rectangle {
4      left: f32,
5      top: f32,
6      width: f32,
7      height: f32,
8      center: Point,
9  }
10
11 impl Rectangle {
12     fn new(left: f32, top: f32, width: f32, height: f32) -> Self
13     {
14         Self {
15             left,
16             top,
```

```

16         width,
17         height,
18         center: Point(left + width / 2.0, top + height /
19                         2.0)
20     }
21
22     fn center(&self) -> &Point {
23         &self.center
24     }
25 }
26
27
28 #[test]
29 fn test() {
30     let rect = Rectangle::new(0.0, 0.0, 100.0, 100.0);
31     match rect.center() {
32         &Point(x, y) => println!("the rectangle center at: ({},
33                                     {})", x, y),
34     }

```

由于 `center` 方法返回的是 `&Point`，我们需要用 `&` 模式来进行匹配。

## 匹配条件守卫

有些时候，我们匹配到某个模式之后，还需要进一步判断和约束，此时可以使用匹配条件守卫（Match Guards）。匹配条件守卫是用于 `match` 语句中的一个额外条件，它们允许你在模式匹配时添加更多的逻辑判断。这些守卫通过在模式后面加上 `if` 关键字和一个布尔表达式来实现，只有当守卫条件为真时，匹配才会成功。

```

1 match some_value {
2     x if x > 10 => println!("大于 10 的值: {}", x),
3     _ => println!("其他值"),
4 }

```

## 匹配多种可能

如果希望在一个分支匹配多种可能的值，可以使用 `part1 | part2` 的方式（这里的 `|` 不是按位或操作符，它更像是正则表达式中的“或”）。

```

1 let at_end = match chars.peek() {
2     Some(& '\r' | & '\n') | None => true,
3     _ => false,
4 };

```

除了可以使用 `|` 匹配多个可能的模式，还可以使用 `..` 匹配一个范围的值：

```

1 match next_char {
2     '0'..='9' => self.read_number(),
3     'a'..='z' | 'A'..='Z' => self.read_word(),
4     ' ' | '\t' | '\n' => self.skip_whitespace(),
5     _ => self.handle_punctuation(),
6 }

```

## 使用 `@` 绑定变量

当我们匹配多种可能或者范围值的时候，我们不知道具体匹配到的是哪个值，从上面的例子可以看出。此时可以使用 `@` 来将匹配到的值绑定到一个变量：

```

1 fn main() {
2     let s = "alkasd7890anjas^fg";
3     let mut iter = s.chars();
4     loop {
5         match iter.next() {
6             Some(n @ '0'..='9') => println!("found a numeric
char: {}", n),
7             Some(c @ ('a' ..= 'z' | 'A' ..= 'Z')) => {
8                 println!("found an alphabet char: {}", c);
9             },
10            Some(c) => println!("found char not a numeric nor
alphabet: {}", c),
11            None => {

```

```

12         println!("End of iteration");
13     break;
14 }
15 }
16 }
17 }
```

在上面的例子中，第一个分支，如果匹配到 `'0' ..= '9'`，那么将匹配到的字符存储到变量 `n`，然后在分支中的语句或者表达式使用这个变量。第二个分支也是类似的情况。

Rust 的模式匹配是非常强大的，但是如果以前没有接触过类似的语法的话，在写代码的时候很难一下子想到使用 `match` 表达式，需要慢慢适应。

## 速查表

为了方便大家学习，这里整理了一个 Rust 模式匹配的速查表。

模式类型	示例	解释
字面量	<code>100</code> <code>"name"</code> <code>DEFAULT_FONT_SIZE</code>	匹配精确的值，这个值可以是字面量，也可以是一个 <code>const</code> 变量名
范围	<code>0 ..= 100</code> <code>'a' ..= 'z'</code> <code>256 ..</code>	匹配一个范围的值
通配符	<code>_</code>	匹配前面分支没有匹配到的值，并忽略它
变量	<code>name</code> <code>mut count</code>	将匹配到的值拷贝（实现了 <code>Copy trait</code> 的） 或者移动到对应的变量中
<code>ref</code> 变量	<code>ref field</code> <code>ref mut field</code>	创建指向匹配到的值的引用 并将其赋值给对应的变量
绑定变量	<code>val @ 0..= 99</code> <code>ref circle @ Shape::Circle</code> <code>{ .. }</code>	如果匹配 <code>@</code> 右侧的模式成功，则将值或者引用移动/拷贝到 <code>@</code> 左侧的变量中

枚举模式	<code>Some(value)</code>
	<code>None</code>
	<code>Pet::Orca</code>
元组模式	<code>(key, value)</code>
	<code>(r, g, b)</code>
数组模式	<code>[first, second]</code>
	<code>[first, _, third]</code>
	<code>[first, .., nth]</code>
	<code>[]</code>
结构体模 式	<code>Color(r, g, b)</code>
	<code>Point {x, y}</code>
	<code>Card { suit: Clubs, rank: n }</code>
	<code>Account {id, name}</code>
引用	<code>&amp;value</code>
	<code>&amp;(k, v)</code>
“或”模式	<code>'a'   'A'</code>
	<code>Some("left"   "right")</code>
条件守卫	<code>x if x * x &lt; r2</code>