13模式和解构

2020年3月9日 12:23

在Rust中，模式有如下一些内容组合而成：
1、字面值
2、解构的数组、枚举、结构体，元组结构体，元组。
3、变量
4、通配符（..省略匹配）
5、占位符（_占位符）

所有可能用到模式的位置：
1、match分支
2、if let 条件表达式
3、while let条件循环
4、 for循环
5、 let
6、函数参数
7、模式匹配分为refutable（可反驳的）和irrefutable（不可反驳的）
　　let语句、函数参数、for循环都是irrefutable（不可反驳的），一定要匹配到，否到编译错误。
　　if let和while let是refutable（可反驳的），可能会匹配不到。

模式的所有语法：
1、匹配字面值
```
match x {
    1 => println!(),
    2 => println!(),
    _ => println!(),
}
```
2、匹配命名变量
```
let x = Some(5);
match x {
    Some(50) => println!(),
    Some(y) => println!(),
    _ => println!(),
}
```

3、多个模式
```
let x = 1;
match x {
    1 | 2 => println!(), //或的意思
    3 => println!(),
    _ => println!(),
}
```
4、通过(..=)匹配值的范围,新版本已经不再建议使用...
　　范围只能用于数字值和char值。
```
let x = 5;
match x {
    1 ..= 5 => println!(), // 1 |2| 3| 4| 5
    - => println!(),
}
```

5、解构并分解值
　　也可以使用模式来解构结构体，元组结构体，枚举，元组和引用。
　　(1)解构结构体
```
let p = Point {x: 0, y: 7};
```

```
                let Point {x: a, y: b} = p;
                let Point{x, y} = p; // 简写的方式
                // 在部分结构体模式中使用字面值进行解构，用于测试一些字段
                 // 为特定值的同时创建其他字段的变量。
                 let p = Point{x:0, y:7};
                 match p {
                     Point{x, y: 0} => println!(),// 匹配y为0时，x的值,目前只支持整数数值和原生
类                  型
                     Point {x: 0, y} => println!(),//x为0时,目前只支持整数数值或者原生类型
                     Point {x, y} => println!(), //全部
                 }
```

(2)解构枚举

使用match

(3)解构引用

(4)解构元组结构体

```
struct Nihao(String);
let n = Nihao(String::from("hades"));
let Nihao(name) = n;
```

6、忽略模式中的值(_和..)

(1)使用_忽略整个值，用在match的最后一个分支，也可以用于任意模式，包括函数参数。

在函数中使用_忽略参数，在实现trait时特别有用，你需要特定类型签名，但是函数实现并不需要某个参数时。

(2)通过在名字前加上一个_来忽略未使用的变量

通过在名字前加上一个_，编译器不会再显示警告未使用的变量了。

只使用_和使用_开头的名称是不同的。_x仍然会将值绑定到变量，_则完全不会绑定变量。

(3)用..忽略剩余的值

```
        let Point{x, ..} = p;
        let (first, .., last) = (2, 4, 5, 16, 32);
        let (.., second, ..) = (2, 3, 4, 5, 6, 54); //error,这里是有歧义的
```

7、匹配守卫（提供了额外的条件）

```
let num = Some(4);
match num {
    Some(x) if x < 5 => println!(),
    Some(x) => println!(),
    None => println!(),
}

let x = 4;
let y = true;
match x {
    1 ..= 10 if y => println!(),
    - => println!(),
}

let x = 3;
let y = true;
match x {
    1 | 2 | 3 if y => println!(), // 等同于(1 | 2 | 3) if y
    - => println!(),
}
```

8、@绑定

```
let x = Some(2);
match x {
    Some(v @ 1 ..= 10) => println!("{}", x),
    Some(_) => println!(),
    None => println!(),
}
```

9、遗留的模式：ref和ref mut

ref和ref mut的出现是为了防止数据所有权移动的。

```
let r = &Some(String::from("hades"));
match r {
    &Some(name) => println!("{}", name),//在老版本的rust中，name字段发生了所有权的移动
    &Some(ref name) => println!("{}", name),// 在老版本中，要这样做，得到一个引用
    &Some(ref mut name) => println!("{}", name),// 如果 r 是一个 &mut Some的话，可以得到一个可改变引用
    None => (),
}
```

但是在今天的版本中，如果match后面的是&借用的，所有创建的绑定都是借用，如果match后面是可改变借用的话，

那么所有创建的绑定都是可改变借用。

但是ref和ref mut依然有价值，因为match后面是可变借用的话，我们可以通过ref改变为不可变借用.

```
let mut a = Some(String::from("hades"));
match &mut a {
    Some(name) => println!(), // name是一个&mut String类型
    Some(ref name) => println!(), //此时name是一个 &String类型了。我觉得也就这一个用处了吧
}
```