泛型

函数中的泛型

结构体中的泛型

枚举的泛型

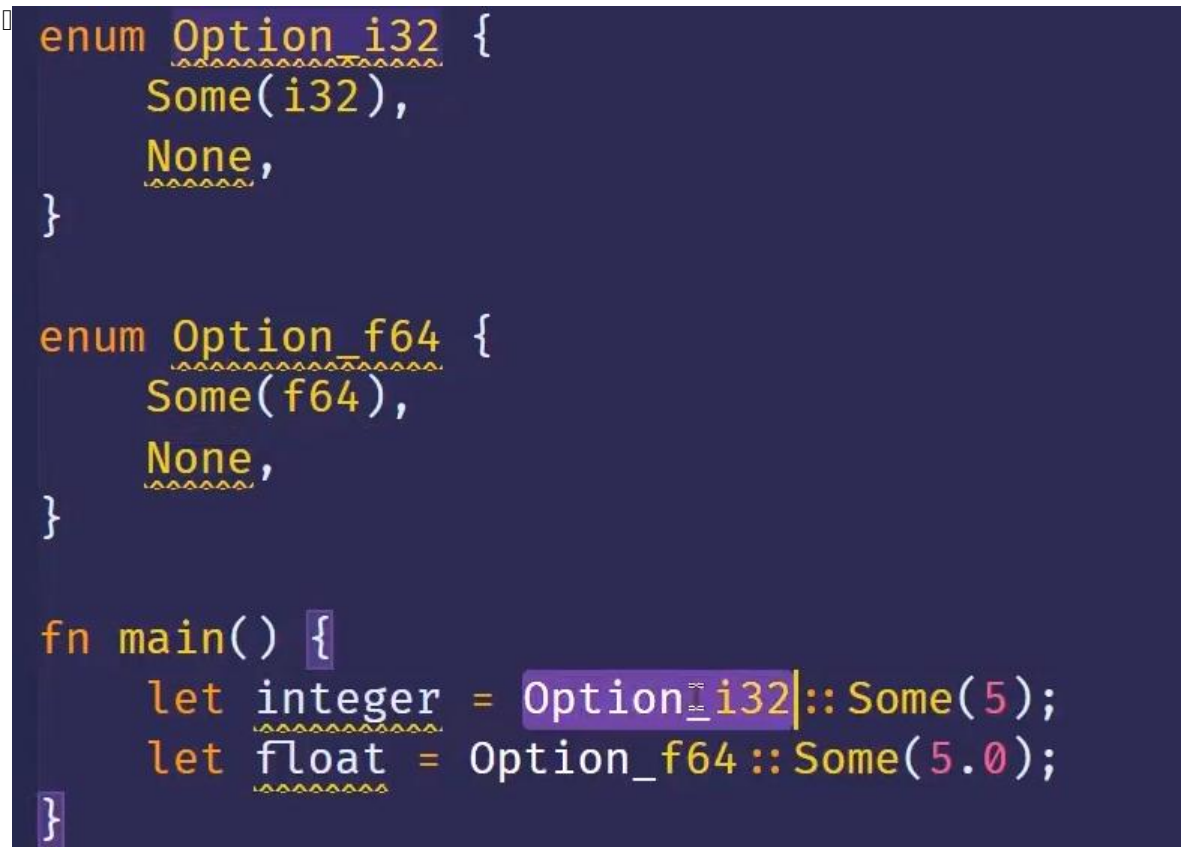
方法 impl 中的泛型

```rust
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 4 };
    let p2 = Point { x: "Hello", y: 'c' };
    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

单态化


enum Option_i32 {
Some(i32),
None,
}
enum Option_f64 {
Some(f64),
None,
}
fn main() {
let integer = Option_i32::Some(5);
let float = Option_f64::Some(5.0);
}

Trait

默认实现

在函数/方法中使用trait

★ 生命周期

每个引用都有自己的生命周期　　有效的作用空间　　大多情况下是隐式存在的 可推断　　生命周期可能以不同方式互相关联的时候需要手动标注生命周期

悬垂引用

作用域

借用检查器用判断被赋值的变量生命周期是否比赋值的值（右边的值）短，否则会发生上例的错误。

泛型的生命周期

当一个函数返回类型包含一个借用值的时候函数的签名而且这个函数的返回值可能会出现多个不同的用，也就是体现不出来返回的是哪一个参数的时候，我们就需要给这个函数的签名加上一个生命周期数了。

生命周期标注语法

函数comper 的 'a 表示传入参数的两个变量中最短的那个生命周期

也就是我们可以说我们在方法签名定义的时候定义的'a，它的生命周期取决于该方法的参数中(这个参需要'a进行标识)最短命的那个生命周期并将这个生命周期交给我们的返回值（如果返回值有生命周期话）。

深入生命周期

在结构中使用引用

生命周期的省略

- 在 Rust 引用分析中所编入的模式称为生命周期省略规则。
  - 这些规则无需开发者来遵守
  - 它们是一些特殊情况，由编译器来考虑
  - 如果你的代码符合这些情况，那么就无需显式标注生命周期
- 生命周期省略规则不会提供完整的推断：
  - 如果应用规则后，引用的生命周期仍然模糊不清 → 编译错误
  - 解决办法：添加生命周期标注，表明引用间的相互关系

一个

方法的生命周期

'static 生命周期