

11 范型

范型

Rust 中的范型（Generic）和 TypeScript 中的范型是相同的概念，用来定义可以适用于各种类型值的类型。例如，如果使用 TypeScript 实现一个类似 Rust 中 `Box<T>` 的类，这个类型可以把各种其他类型的值包装进去。

TypeScript 代码

```
1  class Box<T> {
2      private _content: T;
3
4      constructor(content: T) {
5          this._content = content;
6      }
7
8      public getContent(): T {
9          return this._content;
10     }
11
12     public setContent(content: T): void {
13         this._content = content;
14     }
15 }
16
17 // 使用泛型类
18 const numberBox = new Box<number>(123);
19 console.log(numberBox.getContent()); // 输出: 123
20
21 numberBox.setContent(456);
22 console.log(numberBox.getContent()); // 输出: 456
23
```

```

24 const stringBox = new Box<string>("Hello, TypeScript");
25 console.log(stringBox.getContent()); // 输出: Hello, TypeScript

```

其实我们在前面的内容中，已经多次用到范型了，例如：`Rc<T>`、`Box<T>` 等。我们再看一个 Rust 的例子：假如说，我要在一组 `u32` 的数值中找到最大的那个，我需要这样写：

```

1 fn largest_u32(numbers: &[u32]) -> &u32 {
2     let mut largest = &numbers[0];
3     for n in numbers {
4         if n > largest {
5             largest = n;
6         }
7     }
8     largest
9 }

```

那么如果随着项目的变化，我又需要一个在一组 `u64` 值中找到最大的那个值的函数，我可以复制上面的代码，然后修改入参和返回值的类型：

```

1 fn largest_u64(numbers: &[u64]) -> &u64 {
2     let mut largest = &numbers[0];
3     for n in numbers {
4         if n > largest {
5             largest = n;
6         }
7     }
8     largest
9 }

```

可以看到，除了函数名、入参类型和返回值，其他的逻辑都是一样的。这种情况下我们可以使用范型来使得一段代码逻辑可以同时支持 `u32` 和 `u64` 类型的数值。很容易想到，代码可能是下面这样的：

```

1 // 以下代码无法通过编译
2 fn largest<T>(items: &[T]) -> &T {
3     let mut largest = &items[0];
4     for n in items {
5         if n > largest {
6             largest = n;
7         }
8     }
9
10    largest
11 }
```

其实不然，在编译上面这段代码的时候，会有错误：

```

1 error[E0369]: binary operation `>` cannot be applied to type
`&T`
2 --> temp/src/temp5.rs:26:14
3 |
4 26 |         if n > largest {
5 |             - ^ ----- &T
6 |             |
7 |             &T
8 |
9 help: consider restricting type parameter `T`
10 |
11 23 | fn largest<T: std::cmp::PartialOrd>(items: &[T]) -> &T {
12 |             ++++++ ++++++ ++++++
```

为什么会有这样的报错呢？这是因为，Rust 无法确定范型参数 `T`，是否支持行 5 代码中的 `>` 运算符，所以它不让这段代码通过编译。需要我们明确范型参数 `T` 的边界。感谢 Rust 的编译器，不仅仅给我报错了，还告诉我们应该怎么修改。那我们就按照它的提示修改一下代码，要求范型参数 `T` 是实现了 `std::cmp::PartialOrd` trait 的：

```

1 fn largest<T: std::cmp::PartialOrd>(items: &[T]) -> &T {
2     let mut largest = &items[0];
3     for n in items {
4         if n > largest {
5             largest = n;
6         }
7     }
8
9     largest
10 }
```

这次就可以成功通过编译了。这也是我们在上一次课中提到的，trait 和范型总是紧密结合的，可以用来做边界检测。

我们来测试一下上面的代码：

```

1 fn test() {
2     let n_items = [1u32, 100u32, 34u32, 28u32];
3     let largest_n = largest(&n_items);
4     println!("largest number is: {}", largest_n);
5
6     let f_items = [1.23, 2.34, 3.14, 199.32];
7     let largest_f = largest(&f_items);
8     println!("largest float point number is: {}", largest_f);
9 }
```

从上面的代码可以看出来，调用带有范型的函数的时候，不用指定具体的类型，Rust 会根据输入参数类型或者接收返回值的变量定义类型来推断真实调用过程中的 T 是什么类型。如果类型推断不能正常工作的时候，需要我们显式的指定类型。回顾一下我们在猜数字游戏的代码中，用到过 `str.parse()` 方法来把一个字符串转换成需要的数值，这个方法的定义如下：

```

1 pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err>
2 where
3     F: FromStr,
```

可见，`parse()` 方法是一个范型方法，只要是实现了 `FromStr` trait 的类型都可以。这里我们看到了范型参数 `T` 的 trait 边界的第二种写法：`where`。如果一个函数或者类型有多个范型参数，都写到参数列表中会显得很啰嗦，所以可以单独将范型参数的 trait 边界放到 `where` 中。它的格式如下：

```

1 fn fn_name<T1, T2>(arg1: T1, arg2: T2) -> ReturnType
2 where
3     T1: TraitBoundsForT1,
4     T2: TraitBoundsForT2
5 {
6
7 }
```

为了能够让 Rust 自动推断类型，一般会这样写：

```

1 let n: i32 = "123".parse().unwrap();
2 let f: f32 = "321.45".parse().unwrap();
```

因为我们定义了 `n` 和 `f` 的类型，所以 Rust 会自动推断。如果我们不显式的指定 `n` 和 `f` 的类型呢？就需要这样写：

```

1 let n = "123".parse::<i32>().unwrap();
2 let f = "321.45".parse::<f32>().unwrap();
```

上面看到了一个奇怪的符号：`::<>`，这个是 Rust 独有的一个词，叫做“turbofish”（涡轮鱼？）。

带有范型的枚举类型

带有范型参数的枚举类型在 Rust 中也是很常见的，例如，我们前面体验代码遇到的 `Option<T>`，它的定义如下：

```

1 pub enum Option<T> {
2     None,
3     Some(T),
4 }
```

后面我们在错误处理一章中会继续介绍 `Option<T>` 枚举。

带有范型的结构体

结构体也自持范型参数，例如集合类型 `Vec<T>`，就是类型 `T` 的集合。接下来，我们设计一个带有范型参数的 `Rectangle` 类型，以适配不同精度的需求，例如，当仅需要整数的时候，`Rectangle` 的宽度和高度是 `u32`，当需要浮点数的时候，`Rectangle` 的宽度和高度是 `f32`。

定义看起来和枚举类型很相似：

```

1 struct Rectangle<T> {
2     width: T,
3     height: T
4 }
```

接下来，我们为 `Rectangle<T>` 增加计算面积的方法：

```

1 // 以下代码无法通过编译
2 impl<T> Rectangle<T> {
3     fn area(&self) -> T {
4         self.width * self.height
5     }
6 }
```

按照我们的设想，类型 `T` 是 `u32` 或者 `f32`，所以 `area()` 方法的返回值应该也是对应的 `u32` 或者 `f32`。但是对于任意的类型 `T`，Rust 不确定这个类型是支持乘法（`*`）运算符号的，所以编译这段代码会报错：

```

1 error[E0369]: cannot multiply `T` by `T`
2   --> generic-demo/src/main.rs:10:20
3   |
4 10 |         self.width * self.height
5 |         ----- ^ ----- T
6 |         |
7 |         T
8 |
9 help: consider restricting type parameter `T`
10 |
11 8 | impl<T: std::ops::Mul<Output = T>> Rectangle<T> {
12 |     ++++++
13
14 For more information about this error, try `rustc --explain E0369`.

```

编译器给出的消息很明确了，为了能够让类型为 `T` 的 `width` 和 `height` 属性支持乘法运算，类型 `T` 需要实现 `std::ops::Mul<Output = T>`。那么我们根据编译器给出的错误信息修改代码：

```

1 // 以下代码无法通过编译
2 impl<T: std::ops::Mul<Output = T>> Rectangle<T> {
3     fn area(&self) -> T {
4         self.width * self.height
5     }
6 }

```

上面行 2 的代码中，`std::ops::Mul<Output = T>`，`Output` 是 `Mul` trait 的关联类型，下一部分讲解关联类型。我们先看一下 `Mul` 的定义：

```

1 pub trait Mul<Rhs = Self> {
2     type Output;
3
4     // Required method
5     fn mul(self, rhs: Rhs) -> Self::Output;
6 }

```

这里的 `Output` 是没有指明具体的类型的，我们通过 `std::ops::Mul<Output = T>` 限制类型 `T` 实现了乘法运算，并且相乘结果还是类型 `T`。

但是，改造后的代码仍然无法通过编译：

```

1 error[E0507]: cannot move out of `self.width` which is behind a
2     shared reference
3
4     --> generic-demo/src/main.rs:10:9
5     |
6     |         self.width * self.height
7     |         ^^^^^^^^^^-----^
8     |         |
9     |         `self.width` moved due to usage in operator
10    |         move occurs because `self.width` has type `T`,
11    |         which does not implement the `Copy` trait
12
13   332 |         fn mul(self, rhs: Rhs) -> Self::Output;
14   |         ^^^^
15
16 error[E0507]: cannot move out of `self.height` which is behind a
17     shared reference
18
19   10 |         self.width * self.height
20   |         ^^^^^^^^^^ move occurs because
21   |         `self.height` has type `T`, which does not implement the `Copy` trait
22
23 For more information about this error, try `rustc --explain E0507`.

```

因为 `Mul` trait 的 `mul` 方法是会消费两个操作数的，导致所有权转移。我们思考一下，假想的类型 `T` 是 `u32` 或 `f32`，他们都是 `Copy` 的，所以我们需要进一步对类型 `T` 加以约束：

```

1 use std::ops::Mul;
2
3 struct Rectangle<T> {
4     width: T,
5     height: T
6 }
7
8 impl<T: std::ops::Mul<Output = T> + Copy> Rectangle<T> {
9     fn area(&self) -> T {
10         self.width * self.height
11     }
12 }
```

此时，代码可以通过编译了。我们可以测试一下：

```

1 fn main() {
2     let rect = Rectangle::<u32> {
3         width: 100,
4         height: 200,
5     };
6
7     println!("area of rectangle: {}", rect.area());
8
9     let rect1: Rectangle<f32> = Rectangle {
10         width: 10.0,
11         height: 20.0
12     };
13
14     println!("area of rectangle: {}", rect1.area());
15 }
```

通过上面的代码，我们可以总结出来，范型和 trait 经常一起使用，来约束类型 `T` 需要具备的特征。甚至在 Rust 里面，针对不同的类型 `T` 还可以提供不同的方法，例如，我们定义一个类型，并将其封装到 `Cell<T>` 里面：

```

1 struct Point {
2     x: u32,
3     y: u32,
4 }
5
6 fn main() {
7     let cell_point = Cell::new(Point {
8         x: 0,
9         y: 0
10    });
11     cell_point.take(); // 这句代码无法通过编译
12 }
```

我们来看看 `Cell<T>.take()` 方法:

```

1 impl<T> Cell<T>
2 where
3     T: Default
4 {
5     pub fn take(&self) -> T
6 }
```

这个方法的注释是: 将 `Cell` 中的值取出, 并使用类型 `T` 的默认值填充 `Cell`。所以, 它要求类型 `T` 实现 `Default trait`。

由于 `u32` 是实现了 `Default` 的, 所以我们只需给 `Point` 类型增加属性宏即可:

```

1 #[derive(Default)]
2 struct Point {
3     x: u32,
4     y: u32,
5 }
6
7 fn main() {
8     let cell_point = Cell::new(Point {
9         x: 0,
10        y: 0
11    });
12     cell_point.take();
13 }
```

回想一下我们讲解结构体一章内容的时候，说到结构体可以有多个 `impl` 块。除了可以实现协同开发之外，多个 `impl` 块更多的作用是针对不同的 trait 实现不同的方法。

关联类型

在 Rust 中，还有一个和范型特别相似的概念，叫做关联类型：Associated Type。例如，`TryFrom` 这个 trait，是既有范型又有关联类型的：

```

1 pub trait TryFrom<T>: Sized {
2     type Error;
3
4     // Required method
5     fn try_from(value: T) -> Result<Self, Self::Error>;
6 }
```

当我们给一个类型实现 `TryFrom` trait 的时候，需要明确关联类型 `Error` 的具体类型。例如，下面的代码，我们为元组式结构体 `GreaterThanZero` 实现 `TryFrom<i32>`：

```

1 struct GreaterThanZero(i32);
2
3 impl TryFrom<i32> for GreaterThanZero {
4     type Error = &'static str;
5
6     fn try_from(value: i32) -> Result<Self, Self::Error> {
7         if value <= 0 {
8             Err("GreaterThanZero only accepts values greater
9             than zero!")
10        } else {
11            Ok(GreaterThanZero(value))
12        }
13    }

```

由于 `i32` 是包含了 `0` 和负数的，所以某些值无法成功转换成 `GreaterThanZero` 类型，所
以为 `GreaterThanZero` 类型实现 `TryFrom` 的时候，需要处理无法成功转换的场景，此时
我们需要指定 `TryFrom` trait 中的关联类型 `Error` 的具体类型，在上面的示例中是 `&'static str` 类型。

关联类型和泛型参数都可以为类型提供抽象能力，但它们在 Rust 中的用途和语法上有所区
别，导致关联类型无法简单地用另一个泛型参数代替。

1. 语义上的区别：

- 关联类型：表达的是一种“是什么”的关系。它强调的是与 trait 本身强相关的类
型，例如 `Iterator` trait 中的 `Item` 关联类型，它表明了迭代器所产生的值的类
型。这种关系在 trait 定义时就已确定，而不是在实例化时才确定。
- 泛型参数：表达的是一种“对于任何...”的关系。它强调的是一种通用的算法或数
据结构，可以应用于各种类型，例如 `Vec<T>` 中的 `T`，它表示向量可以存储任何类
型的值。泛型参数在实例化时才被具体确定。

2. 用法上的限制：

- 关联类型：通常用于表达 trait 与其关联类型之间一对一的关系。例如，一个 `Iterator` 只能有一个 `Item` 类型。
- 泛型参数：可以有多个，用于表达更复杂的类型关系。例如，一个函数可以接受多个不同类型的参数。

3. 语法上的区别

- 关联类型：使用 `type` 关键字定义，例如

```
1 trait Iterator {  
2     type Item;  
3     // ...  
4 }
```

- 范型参数：使用 `< >` 定义，例如：

```
1 struct Vec<T> {  
2     // ...  
3 }
```