

10 Trait

Trait 的基础概念

在面向对象的编程语言中，有一个很重要的概念：多态。Rust 中也有对应的实现，那就是 Trait 和范型（Generic）。Trait 接近于 TypeScript 中的虚拟类，但是 JavaScript 中没有对应的特性。例如，使用 TypeScript 定义一个 `Shape` 虚拟类，然后针对圆形、矩形都可以继承这个虚拟类，实现自己的计算逻辑。

TypeScript 代码

```
1 abstract class Shape {
2     abstract area(): number;
3     abstract perimeter(): number;
4 }
5
6 class Rectangle extends Shape {
7     #width: number;
8     #height: number;
9
10    constructor(width: number, height: number) {
11        super();
12        this.#width = width;
13        this.#height = height;
14    }
15
16    area(): number {
17        return this.#width * this.#height;
18    }
19
20    perimeter(): number {
21        return (this.#width + this.#height) * 2;
22    }
}
```

```

23 }
24
25 class Circle extends Shape {
26     #radius: number;
27
28     constructor(r: number) {
29         super();
30         this.#radius = r;
31     }
32
33     area(): number {
34         return this.#radius * this.#radius * Math.PI ;
35     }
36
37     perimeter(): number {
38         return this.#radius * Math.PI * 2;
39     }
40 }
41
42
43 const rect = new Rectangle(100, 100);
44 const circle = new Circle(100);
45
46 console.log(`I'm a rect, my area is: ${rect.area()}`);
47 console.log(`I'm a circle, my area is: ${circle.area()}`);

```

上面的代码翻译成 Rust 就是：

```

1 trait Shape {
2     fn area(&self) -> f32;
3     fn perimeter(&self) -> f32;
4 }
5
6 struct Rectangle {
7     width: f32,
8     height: f32
9 }
10

```

```
11 impl Rectangle {
12     fn new(width: f32, height: f32) -> Self {
13         Self {
14             width,
15             height
16         }
17     }
18 }
19
20 impl Shape for Rectangle {
21     fn area(&self) -> f32 {
22         self.width * self.height
23     }
24
25     fn perimeter(&self) -> f32 {
26         (self.width + self.height) * 2.0
27     }
28 }
29
30 struct Circle {
31     radius: f32,
32 }
33
34 impl Circle {
35     fn new(radius: f32) -> Self {
36         Self {
37             radius
38         }
39     }
40 }
41
42 impl Shape for Circle {
43     fn area(&self) -> f32 {
44         self.radius * self.radius * 3.1415
45     }
46
47     fn perimeter(&self) -> f32 {
48         self.radius * 3.1415 * 2.0
49     }
}
```

```

50 }
51
52 #[test]
53 fn test() {
54     let rect = Rectangle::new(100.0, 100.0);
55     let circle = Circle::new(100.0);
56     println!("I'm a rectangle, my area is: {}", rect.area());
57     println!("I'm a circle, my area is: {}", circle.area());
58 }
```

在上面的 Rust 代码中，我们定义了一个 `trait Shape`，它有两个方法，分别是 `area()` 和 `perimeter()`，这两个方法只是定义了，但是没有对应的实现。然后，我们再定义两个结构体，分别是 `Rectangle` 和 `Circle`，通过一个 `impl` 块来为这两个结构体增加自己的方法，以及实现接口 `Shape` 的方法。前面的课程讲过，给一个结构体添加方法是通过 `impl` 块，那么给一个结构体实现某个接口，写法非常类似：`impl TraitName for StructName {}`。看起来也没有很难，对不对啊？

结合 TypeScript 中的抽象类的知识，以及上面的示例代码，不难得出结论：Rust 中的 `trait` 用来定义结构体的行为。

Trait 方法的默认实现

`Trait` 中的方法不包含方法体代码的，是需要实现这个 `trait` 的类型自行补全方法体的。但是 `trait` 中的方法也可以包含实现代码，这样的方法叫做包含了默认实现的方法。例如：`std::io::Write trait` 的定义（节选）如下：

```

1 pub trait Write {
2     fn write(&mut self, buf: &[u8]) -> Result<usize>;
3     fn flush(&mut self) -> Result<()>;
4     fn write_all(&mut self, mut buf: &[u8]) -> Result<()> {
5         while !buf.is_empty() {
6             match self.write(buf) {
7                 Ok(0) => {
8                     return Err(Error::WRITE_ALL_EOF);
9                 }
10                Ok(n) => buf = &buf[n..],
11            }
12        }
13    }
14 }
```

```

11             Err(ref e) if e.is_interrupted() => {}
12         Err(e) => return Err(e),
13     }
14 }
15 Ok(())
16 }
17 //...
18 }
```

那么我们为一个类型实现 `Write` trait 的时候，只需要实现且必须实现 `write` 和 `flush` 就好了，`write_all` 方法已经有默认实现了。

```

1 /// A Writer that ignores whatever data you write to it.
2 pub struct Sink;
3
4 impl Write for Sink {
5     fn write(&mut self, buf: &[u8]) -> Result<usize> {
6         // Claim to have successfully written the whole buffer.
7         Ok(buf.len())
8     }
9
10    fn flush(&mut self) -> Result<()> {
11        Ok(())
12    }
13 }
14
15 fn main() {
16     let mut out = Sink;
17     out.write_all(b"hello world\n")?;
18 }
```

上面的示例代码中，我们为 `Sink` 结构体实现了 `Write` trait，但是只实现了 `write` 和 `flush` 两个方法，但是我们却可以调用 `write_all` 方法，这种能力就是得益于 trait 方法的默认实现。

常用的 Trait

Rust 标准库中的 trait 分为 3 类：语言扩展 trait、标记性 trait 和普通 trait。

语言扩展 trait 是留给 Rust 扩展我们所编写的类型的，可以让我们自己的类型的使用方式更接近 Rust 语言。

标记性 trait 没有方法，通常用在范型类型边界检查。下节课我们会讲到范型相关的知识。

常见的 trait 有以下几种。

Drop

前面我们在内存管理和所有权章节讲过，当一个值的所有者超出作用范围之后，Rust 会丢弃（drop）这个值。所谓的丢弃值是指释放这个值所占用的内存或者其他系统资源。大部分时候，我们不需要为自己的类型手动实现 Drop trait，Rust 会默默的帮我们完成这一切。当然了，如果你需要的话，可以为自己的结构体实现 `std::ops::Drop` trait，它只有一个方法：

```

1 pub trait Drop {
2     // Required method
3     fn drop(&mut self);
4 }
```

如果你为你的类型手动实现了 Drop，那么当这个类型的值被丢弃的时候，Rust 会调用 `drop()` 方法。需要特别注意的是，在手动为自己的类型实现 Drop trait 的时候，尽量不要在这个方法中创建额外的值。

Deref 和 DerefMut

前面的课程中我们提到过，如果一个变量是一个类型的引用，那么使用 `*` 或者 `.` 操作符就可以完成自动解引用。一般来说，自动解引用 `&T` 的时候，就会得到 `T`。但是回想一下我们在“引用和借用”章节中的一段示例代码：

```

1 fn main() {
2     let mut b = Box::new(5);
3     *b = 6;
4     println!("b = {}", b);
5 }
```

明明 `b` 是一个 `Box<i32>` 类型，为什么 `*b` 可以直接操作 `i32` 的值，而不是 `Box<i32>` 的值？其秘密就在于 `Box<T>` 实现了 `std::ops::DerefMut` trait，同时，它也实现了 `std::ops::Defef` trait，这两个 trait 的定义如下：

```

1 pub trait Deref {
2     type Target: ?Sized;
3
4     // Required method
5     fn deref(&self) -> &Self::Target;
6 }
7
8 pub trait DerefMut: Deref {
9     // Required method
10    fn deref_mut(&mut self) -> &mut Self::Target;
11 }
```

简单来说，这两个 trait 就是可以自定义对于某个类型使用 `*` 或者 `.` 操作符的时候的行为。

Sized

`Sized` 的全称是 `std::marker::Sized`，从这个名字可以看出来，他是一个标记型 trait，不包含任何方法。标记为 `Sized` 的类型表示无论创建的示例的具体值是多少，他们在内存中的长度都是一样大小的。在 Rust 中，绝大部分的类型都是 `Sized` 的，而且，我们不可以给自己编写的结构体或者枚举增加 `Sized` trait。针对我们自己编写的类型，Rust 会自动判断是否是 `Sized`。更多的时候，它是用来进行类型约束的。

Copy 和 Clone

前面我们已经多次提到 `Copy` trait 了，简单来说，就是针对编译时已知大小的类型，都是 `Copy` 的，也就是可拷贝的，例如：整数、浮点数、布尔值等。如果一个数组或者元组中的元素都是 `Copy` 的，那么这个数组或者元组也是 `Copy` 的。如果一个结构体的所有属性都是可拷贝的，那么我们可以在这个结构体上应用属性宏 `#[derive(Copy)]` 来让这个结构体也成为可拷贝的。例如：

```
1 #[derive(Copy, Clone)]
2 struct Point(f32, f32);
```

`Clone` 我们也已经接触过了，它表示将某个值，包括在堆上的值，原封不动的克隆一份。一个类型如果是 `Copy` 的，那么它一定是 `Clone` 的，反之则不成立。所以我们上面的例子中，使用属性宏 `#[derive(Copy, Clone)]` 给这个类型实现了 `Copy` 和 `Clone`。

`Copy` 是编译器决定的，我们不可以为非 `Copy` 的类型自己实现 `Copy`。

`Copy` 和 `Drop` 是互斥的，也就是说，一个类型如果是 `Copy` 的，那么它就不是 `Drop` 的。

Default

很多时候，一个结构体有默认值是一种合理的存在。如果一个类型的所有的属性都是实现了 `Default` 的，那么我们可以通过属性宏 `#[derive(Default)]` 来给这个类型实现 `Default`。否则，我们就需要使用 `impl Default for StructName {}` 来给这个类型实现 `Default`。

例如，我可以这样做：

```
1 #[derive(Copy, Clone, Default)]
2 struct Point(f32, f32);
```

也可以：

```

1 struct Point {
2     x: f32,
3     y: f32
4 };
5
6 impl Default for Point {
7     fn default() -> Self {
8         Self {
9             x: 0.0,
10            y: 0.0
11        }
12    }
13 }
```

`Default` 的实战应用场景是这样的，当一个结构体的属性太多的时候，我们可以借助 `Default` 来填充一部分属性的默认值。加入说，我在编写一个应用配置的结构体，其中有一个是数据库连接的配置，代码如下：

```

1 struct ConnectionConfig {
2     host: String,
3     port: u16,
4     db: String,
5     user: String,
6     password: String,
7     character_set: String,
8     min_pool_size: u16,
9     max_pool_size: u16,
10    timeout_seconds: u16,
11    test_on_connected: bool,
12    debug: bool
13 }
```

这么多的属性，如果每次创建这个结构体的实例都需要写这么多，太麻烦了。所以，我们可以给它实现 `Default`：

```
1 impl Default for ConnectionConfig {
```

```

2     fn default() -> Self {
3         Self {
4             host: "127.0.0.1".to_string(),
5             port: 3306,
6             db: "test".to_string(),
7             user: "test".to_string(),
8             password: "password".to_string(),
9             character_set: "utf8".to_string(),
10            min_pool_size: 1,
11            max_pool_size: 200,
12            timeout_seconds: 300,
13            test_on_connected: false,
14            debug: false,
15        }
16    }
17 }
```

然后，再提供一个方法，支持少量属性值即可生成对应实例：

```

1 impl ConnectionConfig {
2     fn new(host: &str, db: &str, user: &str, pwd: &str) -> Self
3     {
4         Self {
5             host: host.to_string(),
6             db: db.to_string(),
7             user: user.to_string(),
8             password: pwd.to_string(),
9             ..Default::default()
10        }
11    }
```

上面这段代码的行 8，表示使用默认值填充其他没有指定的属性。很方便，对吧。

AsRef 和 AsMut

如果一个类型 `S` 实现了 `AsRef<T>`，那么我们可以直接从类型 `S` 获得 `&T`。例如：`Vec<T>` 实现了 `AsRef<[T]>`，所以我们可以直接从 `Vec<T>` 获得 `&[T]`，`String` 和 `str` 实现了 `AsRef<Path>`，所以我们可以直接从 `String` 得到 `&Path`。

例如：

```

1 fn test() {
2     let _ = check_file("/path/to/the/file.ext");
3     let s = "/path/to/the/file.ext".to_string();
4     let _ = check_file(s);
5 }
6
7 fn check_file<P: AsRef<Path>>(path: P) -> Result<(), io::Error>{
8     Ok(())
9 }
```

上面示例代码 `check_file()` 方法接受 `AsRef<Path>` 类型的参数，但是实际调用的时候，我们既可以传入 `String`，又可以传入 `&str`（行 2 代码）。但是，明明前面说的是 `str` 类型实现了 `AsRef<Path>`，为什么传入 `&str` 也可以呢？这是因为在编译的时候，标准库帮我们做了这么一件事情：对于类型 `T` 和 `U`，如果 `T: AsRef<U>`，那么 `&T: AsRef<U>`。

`AsMut` 顾名思义，和 `AsRef` 类似，但是可以得到一个值的可变引用。

如果我们在编写一些通用的库给别人用的时候，或者在项目中的公共代码给其他模块用的时候，定义函数参数的时候，建议使用 `AsRef` 指定参数类型，方便调用者使用。

From 和 Into

`std::convert::From` 和 `std::convert::Into` 用来将一个类型的值转换成另外一个类型的值。他们的定义如下：

```

1 pub trait From<T>: Sized {
2     // Required method
3     fn from(value: T) -> Self;
4 }
5
6 pub trait Into<T>: Sized {
7     // Required method
8     fn into(self) -> T;
9 }

```

从这两个 trait 的定义上可以看出，在调用 `from` 或者 `into` 方法的时候，原有的值被消费，发生了所有权转移。任意的类型 `A`，都会被编译器增加 `From<A>` 和 `Into<A>` 的实现。假定我们有一个自定义类型 `A`，只需给 `A` 实现 `From<T>`，对应的类型 `T` 会被编译器自动增加 `Into<A>` 的实现。

看一个具体的例子，标准库的 `std::net::Ipv4Addr` 结构体，它实现了 `From<Ipv4Addr>`、`From<[u8; 4]>`、`From<u32>`：

The screenshot shows the Rust documentation for the `Ipv4Addr` struct. The sidebar on the left lists various traits implemented by `Ipv4Addr`, with `From<Ipv4Addr>` being highlighted with a red box. The main content area shows the `Struct std::net::Ipv4Addr` with its definition:

```
pub struct Ipv4Addr { /* private fields */ }
```

It also provides a textual representation and examples.

所以，当调用一个需要 `Ipv4Addr` 类型参数的函数时，我们可以使用很自由的形式。例如，定义一个 `ping` 函数，该函数接受 `Into<Ipv4Addr>` 类型的参数：

```

1 use std::net::Ipv4Addr;
2
3 fn ping<A>(address: A) -> std::io::Result<bool>
4     where A: Into<Ipv4Addr>
5 {
6     let ipv4_addr = address.into();
7 }
```

那么当我们调用这个函数的时候，只要传入的参数可以 `Into<Ipv4Addr>`，就可以：

```

1 fn main() {
2     let _ = ping(Ipv4Addr::new(23, 21, 68, 141));
3     let _ = ping([66, 146, 219, 98]);
4     let _ = ping(0xd076eb94_u32);
5 }
```

TryFrom 和 TryInto

有时候，从一个类型的值转换到另外一个类型的值不总是能够成功的，所以 Rust 还提供了 `TryFrom` 和 `TryInto` 这两个 trait。它们的用途和 `From`、`Into` 很相似，区别就是返回值是一个 `Result` 类型，可以处理转换失败的场景。

```

1 pub trait TryFrom<T>: Sized {
2     type Error;
3
4     // Required method
5     fn try_from(value: T) -> Result<Self, Self::Error>;
6 }
7
8 pub trait TryInto<T>: Sized {
9     type Error;
10
11    // Required method
12    fn try_into(self) -> Result<T, Self::Error>;
13 }
```

总结

Rust 虽然没有提供结构体的继承能力，但是它的 trait 设计非常优秀，可以通过不同的组合给类型增加能力。其实在我们真正的开发业务系统或者最终应用的时候，从设计角度讲，应该优先选择组合，而不是继承。因为设计一套完整的类型继承，是需要非常多的前期工作，否则很容易到后期出现问题。使用组合的方式就相对简单一些。