【译】Rust标准库Trait指南（三）

mp.weixin.qq.com/s/Un-VU2_Ysni0CbYvVhEiiA

> 原文标题：Tour of Rust's Standard Library Traits
> 原文链接：https://github.com/pretzelhammer/rust-blog/blob/master/posts/tour-of-rusts-standard-library-traits.md
> 公众号：Rust 碎碎念
> 翻译 by：Praying

内容目录（译注：✅ 表示本文已翻译 ⏰ 表示后续翻译）

泛型 traits

Default

所需预备知识

- Self
- 函数（Functions）
- 派生宏（Derive Macros）

```
trait Default {
    fn default() -> Self;
}
```

可以为实现了 `Default` 的类型构造默认值。

```
struct Color {
    r: u8,
    g: u8,
    b: u8,
}

impl Default for Color {
    // default color is black
    fn default() -> Self {
        Color {
            r: 0,
            g: 0,
            b: 0,
        }
    }
}
```

这在快速构建原型的时候十分有用，尤其是在我们没有过多要求而只需要一个类型实例的情况下：

```
fn main() {
    // just give me some color!
    let color = Color::default();
}
```

当我们想要显式地把函数暴露给用户时，也可以选择这样做：

```
struct Canvas;
enum Shape {
    Circle,
    Rectangle,
}

impl Canvas {
    // let user optionally pass a color
    fn paint(&mut self, shape: Shape, color: Option<Color>) {
        // if no color is passed use the default color
        let color = color.unwrap_or_default();
        // etc
    }
}
```

当我们需要构造泛型类型时， `Default` 在泛型上下文中也是有用的：

```
fn guarantee_length<T: Default>(mut vec: Vec<T>, min_len: usize) -> Vec<T> {
    for _ in 0..min_len.saturating_sub(vec.len()) {
        vec.push(T::default());
    }
    vec
}
```

我们还可以利用 `Default` 类型结合 Rust 的结构体更新语法（struct update syntax）来对结构体部分初始化。现在，我们有一个 `Color` 结构体构造函数 `new` ，该函数接收结构体的所有成员作为参数：

```
impl Color {
    fn new(r: u8, g: u8, b: u8) -> Self {
        Color {
            r,
            g,
            b,
        }
    }
}
```

但是，我们可以有更为便利的构造函数，这些构造函数分别只接收结构体的一部分成员，结构体剩下的其他成员使用默认值：

```
impl Color {
    fn red(r: u8) -> Self {
        Color {
            r,
            ..Color::default()
        }
    }
    fn green(g: u8) -> Self {
        Color {
            g,
            ..Color::default()
        }
    }
    fn blue(b: u8) -> Self {
        Color {
            b,
            ..Color::default()
        }
    }
}
```

还有一个 `Default` 派生宏，通过使用它我们可以像下面这样来写 `Color` ：

```
// default color is still black
// because u8::default() == 0
#[derive(Default)]
struct Color {
    r: u8,
    g: u8,
    b: u8
}
```

Clone

所需预备知识

- Self
- 方法（Methods）
- 默认实现（Default Impls）
- 派生宏（Derive Macros）

```
trait Clone {
fn clone(&self) -> Self;

        // provided default impls
    fn clone_from(&mut self, source: &Self);
}
```

我们能够把 `Clone` 类型的不可变引用转换为所拥有的值，即 `&T -> T` 。 `Clone` 不保证这种转换的效率，所以它会很慢并且成本较高。我们可以使用派生宏在一个类型上快速实现 `Clone` ：

```
#[derive(Clone)]
struct SomeType {
    cloneable_member1: CloneableType1,
    cloneable_member2: CloneableType2,
// etc
}

    // macro generates impl below
impl Clone for SomeType {
    fn clone(&self) -> Self {
        SomeType {
            cloneable_member1: self.cloneable_member1.clone(),
            cloneable_member2: self.cloneable_member2.clone(),
            // etc
        }
    }
}
```

`Clone` 可以用于在泛型上下文中构造一个类型实例。下面是从前面章节拿过来的一个例子，其中的 `Default` 被替换为了 `Clone` ：

```
fn guarantee_length<T: Clone>(mut vec: Vec<T>, min_len: usize, fill_with: &T) -
> Vec<T> {
    for _ in 0..min_len.saturating_sub(vec.len()) {
        vec.push(fill_with.clone());
    }
    vec
}
```

人们通常把克隆（clone）作为一种避免和借用检查器打交道的逃生出口（escape hatch）。管理带有引用的结构体很具有挑战性，但是我们可以通过克隆把引用变为所拥有的值。

```
// oof, we gotta worry about lifetimes 😟
struct SomeStruct<'a> {
    data: &'aVec<u8>,
}

 // now we're on easy street 😎
struct SomeStruct {
    data: Vec<u8>,
}
```

如果我们正在编写的程序对性能不敏感，那么我们就不需要担心克隆数据的问题。Rust 是一门暴露了很多底层细节的语言，所以开发者很容易陷入过早的优化而非真正解决眼前的问题。对于很多程序来讲，最好的优先级顺序通常是，首先构建正确性，其次是优雅性，第三是性能，仅当在对性能进行剖析并确定性能瓶颈之后再去关注性能。通常而言，这是一个值得采纳的好建议，但是你需要清楚，它未必适用于你的程序。

Copy

所需预备知识

- 标记 Trait（Marker Trait）
- Subtraits & SuperTraits
- 派生宏（Derive Macros）

```
trait Copy:Clone{}
```

我们拷贝 `Copy` 类型，例如： `T -> T` . `Copy` 承诺拷贝操作是简单的按位拷贝，所以它是快速高效的。我们不能自己实现 `Copy` ，只有编译器可以提供实现，但是我们可以通过使用 `Copy` 派生宏让编译器这么做，就像使用 `Clone` 派生宏一样，因为 `Copy` 是 `Clone` 的一个 subtrait:

```
#[derive(Copy, Clone)]
struct SomeType;
```

`Copy` 对 `Clone` 进行了细化。一个克隆（clone）操作可能很慢并且开销很大，但是拷贝（copy）操作保证是快速且开销较小的，所以拷贝是一种更快的克隆操作。如果一个类型实现了 `Copy` ， `Clone` 实现就无关紧要了：

```
// this is what the derive macro generates
impl<T: Copy> Clone for T {
    // the clone method becomes just a copy
    fn clone(&self) -> Self {
        *self
    }
}
```

当一个类型实现了 `Copy` 之后，它在被移动（move）时的行为就发生了改变。默认情况下，所有的类型都有*移动（move）语义*，但是一旦某个类型实现了 `Copy` ，它就有了*拷贝（copy）语义*。为了解释二者的不同，让我们看一下这些简单的场景：

```
// a "move", src: !Copy
let dest = src;

// a "copy", src: Copy
let dest = src;
```

在上面两种情况下， `dest = src` 对 `src` 的内容进行按位拷贝并把结果移动到 `dest` ，唯一的不同是，在第一种情况（"a move"）中，借用检查器使得 `src` 变量失效并确保它后面不会在任何其他地方被使用;在第二种情况下（"a copy"）中， `src` 仍然是有效且可用的。

简而言之：拷贝就是移动，移动就是拷贝。它们之间唯一的区别就是其对待借用检查器的方式。

来看一个关于移动（move）的更具体的例子，假定 `sec` 是一个 `Vec<i32>` 类型，并且它的内容看起来像下面这样：

```
{ data: *mut [i32], length: usize, capacity: usize }
```

当我们执行了 `dest = src` ，我们会得到：

```
src = { data: *mut [i32], length: usize, capacity: usize }
dest = { data: *mut [i32], length: usize, capacity: usize }
```

在这个位置， `src` 和 `dest` 对同一份数据各有一个可变引用别名，这是一个大忌，因此，借用检查器让 `src` 变量失效，在编译器不报错的情况下。使得它不能再被使用。

再来看一个关于拷贝（copy）的更具体的例子，假定 `src` 是一个 `Option<i32>` ，且它的内容看起来如下：

```
{ is_valid: bool, data: i32 }
```

现在，当我们执行 `dest = src` 时，我们会得到：

```
src = { is_valid: bool, data: i32 }
dest = { is_valid: bool, data: i32 }
```

它们俩同时都是可用的！因此， `Option<i32>` 是 `Copy` 。

尽管 `Copy` 是一个自动 trait，但是 Rust 语言设计者决定，让类型显式地选择拷贝语义，而不是在类型符合条件时默默地继承拷贝语义，因为后者可能会引起经常导致 bug 的混乱行为。

Any

所需预备知识

- Self
- Generic Blanket Impls
- Subtraits & Supertraits
- Trait Objects

```
trait Any: 'static {
    fn type_id(&self) -> TypeId;
}
```

Rust 的多态风格是参数化的，但是如果我们正在尝试使用一种类似于动态类型语言的更为特别（ad-hoc）的多态风格，那么我们可以通过使用 `Any` trait 来进行模拟。我们不必手动为我们的类型实现 `Any` trait，因为这已经被 generic blanket impl 所涵盖：

```
impl<T: 'static + ?Sized> Any for T {
fn type_id(&self) -> TypeId {
        TypeId::of::<T>()
    }
}
```

我们通过使用 `downcast_ref::<T>()` 和 `downcast_mut::<T>()` 方法从一个 `dyn Any` 中拿出一个 `T`：

```
use std::any::Any;

#[derive(Default)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
fn inc(&mutself) {
self.x += 1;
self.y += 1;
    }
}



fn map_any(mut any: Box<dyn Any>) -> Box<dyn Any> {
ifletSome(num) = any.downcast_mut::<i32>() {
        *num += 1;
    } elseifletSome(string) = any.downcast_mut::<String>() {
        *string += "!";
    } elseifletSome(point) = any.downcast_mut::<Point>() {
        point.inc();
    }
    any
}

            fn main() {
    let mut vec: Vec<Box<dyn Any>> = vec![
        Box::new(0),
        Box::new(String::from("a")),
        Box::new(Point::default()),
    ];
    // vec = [0, "a", Point { x: 0, y: 0 }]
    vec = vec.into_iter().map(map_any).collect();
    // vec = [1, "a!", Point { x: 1, y: 1 }]
}
```

这个 trait 很少需要用到，因为在大多数情况下，参数化多态要优于临时多态性，后者也可以用枚举（enum）来模拟，枚举具有更好的类型安全，需要的间接（抽象）也更少。例如，我们可以用下面的方式实现上面的例子：

```
#[derive(Default)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
fn inc(&mutself) {
self.x += 1;
self.y += 1;
    }
}



enum Stuff {
    Integer(i32),
String(String),
    Point(Point),
}



fn map_stuff(mut stuff: Stuff) -> Stuff {
match &mut stuff {
        Stuff::Integer(num) => *num += 1,
        Stuff::String(string) => *string += "!",
        Stuff::Point(point) => point.inc(),
    }
    stuff
}



fn main() {
letmut vec = vec![
        Stuff::Integer(0),
        Stuff::String(String::from("a")),
        Stuff::Point(Point::default()),
    ];
// vec = [0, "a", Point { x: 0, y: 0 }]
    vec = vec.into_iter().map(map_stuff).collect();
// vec = [1, "a!", Point { x: 1, y: 1 }]
}
```

尽管 `Any` 很少被需要用到，但是在某些时候它也会十分地便利，正如我们在后面错误处理（Error Handling）部分所看到的那样。


Rust碎碎念
长按二维码识别 一键关注新动向
Rust碎碎念