

【译】Rust标准库Trait指南（四）



mp.weixin.qq.com/s/mRSW2xK1NAREp3VikLTGEg

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)

- 引言 
- Trait 基础 
- 自动 Trait 
- 泛型 Trait 
- 格式化 Trait 
- 操作符 Trait  => 
- 转换 Trait 
- 错误处理 
- 迭代器 Trait 
- I/O Trait 
- 总结 

【译】Rust标准库Trait指南（一）

【译】Rust标准库Trait指南（二）

【译】Rust标准库Trait指南（三）

格式化 Traits (Formatting Traits)

我们可以使用 `std::fmt` 中的格式化宏来把类型序列化(serialize)为字符串, 其中最为我们熟知的就是 `println!`。我们可以把格式化参数传递给 `{}` 占位符, 这些占位符用于选择使用哪个 trait 来序列化占位符参数。

Trait	Placeholder	Description
<code>Display</code>	<code>{}</code>	显示表示
<code>Debug</code>	<code>{:?}</code>	调试表示
<code>Octal</code>	<code>{:o}</code>	八进制表示
<code>LowerHex</code>	<code>{:x}</code>	小写十六进制表示
<code>UpperHex</code>	<code>{:X}</code>	大写十六进制表示
<code>Pointer</code>	<code>{:p}</code>	内存地址

Trait	Placeholder	Description
Binary	<code>{:b}</code>	二进制表示
LowerExp	<code>{:e}</code>	小写指数表示
UpperExp	<code>{:E}</code>	大写指数表示

Display & ToString

```
trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}
```


`Display` 类型可以被序列化为对用户更为友好的 `String` 类型。以 `Point` 类型为列：

```
use std::fmt;

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

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}

fn main() {
    println!("origin: {}", Point::default());
    // prints "origin: (0, 0)"

    // get Point's Display representation as a String
    let stringified_point = format!("{}", Point::default());
    assert_eq!("(0, 0)", stringified_point); // 
}
```

除了使用 `format!` 宏让一个类型以 `String` 类型显示，我们还可以使用 `ToString` trait：


```
trait ToString {
    fn to_string(&self) -> String;
}
```


这个 trait 不需要我们实现，事实上，由于 generic blanket impl，我们也不能去实现它，因为所有实现了 `Display` 的类型都会自动实现 `ToString`：


```
impl<T: Display + ?Sized> ToString for T;
```

在 `Point` 上使用 `ToString`：

```

#[test]// 
fn display_point() {
    let origin = Point::default();
    assert_eq!(format!("{}", origin), "(0, 0)");
}

#[test]// 
fn point_to_string() {
    let origin = Point::default();
    assert_eq!(origin.to_string(), "(0, 0)");
}

#[test] // 
fn display_equals_to_string() {
    let origin = Point::default();
    assert_eq!(format!("{}", origin), origin.to_string());
}

```

Debug

```

trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result;
}

```

Debug 和 **Display** 有着相同的签名。唯一的不同在于，只有当我们指定了 **{:?}** 才会调用 **Debug** 实现。 **Debug** 可以被派生：

```

use std::fmt;

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

// derive macro generates impl below
impl fmt::Debug for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.debug_struct("Point")
            .field("x", &self.x)
            .field("y", &self.y)
            .finish()
    }
}

```

为一个类型实现 **Debug** 能够使得这个类型在 **dbg!** 中使用， **dbg!** 宏在快速打印日志方面比 **println!** 更合适，它的一些优势如下：

1. **dbg!** 打印到 **stderr** 而不是 **stdout**，因此在我们的程序中，能够很容易地和标准输出的输出结果区分。
2. **dbg!** 会连同传入的表达式和表达式的计算结果一起打印出来。
3. **dbg!** 会获取传入参数的所有权并将其返回，因此你可以在表达式中使用它：

```
fn some_condition() -> bool {
    true
}

// no logging
fn example() {
    if some_condition() {
        // some code
    }
}

// println! logging
fn example_println() {
    // 🧑
    let result = some_condition();
    println!("{}", result); // just prints "true"
    if result {
        // some code
    }
}

// dbg! logging
fn example_dbg() {
    // 🧑
    if dbg!(some_condition()) { // prints "[src/main.rs:22] some_condition() = true"
        // some code
    }
}
```

dbg! 的唯一缺点就是它不会在 release 构建中自动裁剪，所以如果我们不想在最后生成的二进制包含这些内容，就必须手动移除它。

操作符 Trait（Operator Traits）

Rust 中所有的操作符都和 trait 关联，如果我们想要为我们的类型实现一些操作符，我们就必须实现与之关联的 trait。

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
<code>Eq</code> , <code>PartialEq</code>	比较	<code>==</code>	相等
<code>Ord</code> , <code>PartialOrd</code>	比较	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code>	比较
<code>Add</code>	算术	<code>+</code>	相加

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
AddAssign	算术	+=	相加并赋值
BitAnd	算术	&	按位与
BitAndAssign	算术	&=	按位与并赋值
BitXor	算术	^	按位异或
BitXorAssign	算术	^=	按位异或并赋值
Div	算术	/	除
DivAssign	算术	/=	除并赋值
Mul	算术	*	乘
MulAssign	算术	*=	乘并赋值
Neg	算术	-	一元求反
Not	算术	!	一元逻辑求反
Rem	算术	%	求余
RemAssign	算术	%=	求余并赋值
Shl	算术	<<	左移
ShlAssign	算术	<<=	左移并赋值
Shr	算术	>>	右移
ShrAssign	算术	>>=	右移并赋值
Sub	算术	-	减
SubAssign	算术	-=	减并赋值
Fn	闭包	(...args)	不可变闭包调用
FnMut	闭包	(...args)	可变闭包调用
FnOnce	闭包	(...args)	一次性闭包调用
Deref	其他	*	不可变解引用

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
DerefMut	其他	*	可变解引用
Drop	其他	-	类型析构
Index	其他	[]	不可变索引
IndexMut	其他	[]	可变索引
RangeBounds	其他	..	区间

比较 Trait（Comparison Traits）

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
Eq , PartialEq	比较	==	相等
Ord , PartialOrd	比较	< , > , <= , >=	比较

PartialEq & Eq

```
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;

    // provided default impls
    fn ne(&self, other: &Rhs) -> bool;
}
```

`PartialEq<Rhs>` 类型可以通过 `==` 操作符检查是否和 `Rhs` 类型相等。

所有的 `PartialEq<Rhs>` 实现必须确保相等性是对称的和可传递的。这意味着，对于任意的 `a`、`b`、`c`：

- `a == b` 也意味着 `b == a`（对称性）
- `a == b && b == c` 意味着 `a == c`（传递性）

默认情况下，`Rhs = Self`，因为我们几乎总是想要比较同一类型的不同实例，而不是不同类型的不同实例。这也保证了我们的实现是对称的和可传递的。

```
struct Point {
    x: i32,
    y: i32
}

// Rhs == Self == Point
impl PartialEq for Point {
    // impl automatically symmetric & transitive
    fn eq(&self, other: &Point) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

如果一个类型的所有成员都实现了 `PartialEq`，则它会派生实现 `PartialEq`：

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

#[derive(PartialEq)]
enum Suit {
    Spade,
    Heart,
    Club,
    Diamond,
}
```

一旦我们为自己的类型实现了 `PartialEq`，我们就能够轻松地在类型的引用之间进行相等性比较，这要归功于 generic blanket impls：

```

// this impl only gives us: Point == Point
#[derive(PartialEq)]
struct Point {
    x: i32,
    y: i32
}

// all of the generic blanket impls below
// are provided by the standard library

// this impl gives us: &Point == &Point
impl<A, B> PartialEq<&'_ B> for &'_ A
where A: PartialEq<B> + ?Sized, B: ?Sized;

// this impl gives us: &mut Point == &mut Point
impl<A, B> PartialEq<&'_ mut B> for &'_ mut A
where A: PartialEq<B> + ?Sized, B: ?Sized;

// this impl gives us: &Point == &mut Point
impl<A, B> PartialEq<&'_ mut B> for &'_ A
where A: PartialEq<B> + ?Sized, B: ?Sized;

// this impl gives us: &mut Point == &mut Point
impl<A, B> PartialEq<&'_ mut B> for &'_ mut A
where A: PartialEq<B> + ?Sized, B: ?Sized;

```

因为这个 trait 是泛型的，所以我们可以不同的类型之间定义相等性（比较）。标准库利用这一点实现了类字符串类型之间的相互比较，比如 `String`、`&str`、`PathBuf`、`&Path`、`OsString`、`&OsStr` 等等。

通常，我们应该仅为特定的不同类型之间实现相等性，这些不同类型包含了相同类型的数据，并且它们之间唯一的区别是表现数据的方式和与数据交互的方式。

下面是一个反面实例，关于某人试图在没有满足上述规则的不同类型之间实现 `PartialEq` 用以检查完整性的例子：


```
#[derive(PartialEq)]
enum Suit {
    Spade,
    Club,
    Heart,
    Diamond,
}

#[derive(PartialEq)]
enum Rank {
    Ace,
    Two,
    Three,
    Four,
    Five,
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King,
}

#[derive(PartialEq)]
struct Card {
    suit: Suit,
    rank: Rank,
}

// check equality of Card's suit
implPartialEq<Suit> for Card {
    fn eq(&self, other: &Suit) -> bool {
        self.suit == *other
    }
}

// check equality of Card's rank
implPartialEq<Rank> for Card {
    fn eq(&self, other: &Rank) -> bool {
        self.rank == *other
    }
}

fn main() {
    let AceOfSpades = Card {
        suit: Suit::Spade,
        rank: Rank::Ace,
    };
    assert!(AceOfSpades == Suit::Spade); // ✓
    assert!(AceOfSpades == Rank::Ace); // ✓
}
```

`Eq` 是一个标记 trait，并且是 `PartialEq<Self>` 的一个 subtrait。

```
trait Eq: PartialEq<Self> {}
```

如果我们为一个类型实现了 `Eq`，在 `PartialEq` 所要求的对称性和可传递性之上，我们还保证了反射性（reflexivity），也就是对于任意的 `a`，都有 `a == a`。从这种意义上来说，`Eq` 对 `PartialEq` 进行了细化，因为它表示了一个更为严格的相等性。如果一个类型的所有成员都实现了 `Eq`，那么 `Eq` 的实现可以派生到这个类型。

浮点型实现了 `PartialEq` 但是没有实现 `Eq`，因为 `NaN != NaN`。几乎所有其他的实现了 `PartialEq` 的类型都实现了 `Eq`，除非它们包含浮点类型。

一旦一个类型实现了 `PartialEq` 和 `Debug`，我们就可以可以在 `assert_eq!` 宏中使用它。我们还可以比较实现了 `PartialEq` 类型的集合。

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

fn example_assert(p1: Point, p2: Point) {
    assert_eq!(p1, p2);
}

fn example_compare_collections<T: PartialEq>(vec1: Vec<T>, vec2: Vec<T>) {
    // if T: PartialEq this now works!
    if vec1 == vec2 {
        // some code
    } else {
        // other code
    }
}
```

Hash

```
trait Hash {
    fn hash<H: Hasher>(&self, state: &mut H);

    // provided default impls
    fn hash_slice<H: Hasher>(data: &[Self], state: &mut H);
}
```

这个 trait 没有与任何操作符关联，但是讨论它的最好时机就是在 `PartialEq` 和 `Eq` 之后，所以把它写在这里。`Hash` 类型可以通过一个 `Hasher` 被（计算）哈希。

```

use std::hash::Hasher;
use std::hash::Hash;

struct Point {
    x: i32,
    y: i32,
}

impl Hash for Point {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        hasher.write_i32(self.x);
        hasher.write_i32(self.y);
    }
}

```

使用派生宏可以生成和上面一样的实现：

```

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

```

如果一个类型同时实现了 **Hash** 和 **Eq**，那么这些实现必须达成一致，从而保证对于所有的 **a** 和 **b**，如果 **a == b** 那么 **a.hash() == b.hash()**。因此，当为一个类型同时实现这两个 trait 时，要么都用派生宏，要么都手动实现，但是不要混合，否则我们就有可能破坏上面的不变性。

为一个类型实现 **Eq** 和 **Hash** 的最大好处是，它让我们能够把类型作为 key 存储在 **HashMap** 和 **HashSet** 中。

```

use std::collections::HashSet;

// now our type can be stored
// in HashSets and HashMaps!
#[derive(PartialEq, Eq, Hash)]
struct Point {
    x: i32,
    y: i32,
}

fn example_hashset() {
    letmut points = HashSet::new();
    points.insert(Point { x: 0, y: 0 }); // ✅
}

```

PartialOrd & Ord

```

enum Ordering {
    Less,
    Equal,
    Greater,
}

trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    // provided default impls
    fn lt(&self, other: &Rhs) -> bool;
    fn le(&self, other: &Rhs) -> bool;
    fn gt(&self, other: &Rhs) -> bool;
    fn ge(&self, other: &Rhs) -> bool;
}

```

`PartialOrd<Rhs>` 类型可以通过 `<`、`<=`、`>=` 操作符和 `Rhs` 类型比较。所有的 `PartialOrd<Rhs>` 实现必须保证比较时非对称和可传递的。这意味着，对于任意的 `a`、`b` 和 `c`：

- `a < b` 意味着 `!(a > b)`（非对称性）
- `a < b && b < c` 意味着 `a < c`（传递性）

`PartialOrd` 是 `PartialEq` 的一个 subtrait，并且它们的实现必须相互一致。

```

fn must_always_agree<T: PartialOrd + PartialEq>(t1: T, t2: T) {
    assert_eq!(t1.partial_cmp(&t2) == Some(Ordering::Equal), t1 == t2);
}

```

当比较 `PartialEq` 类型时，我们可以检查是否它们相等或者不相等，但是当比较 `PartialOrd` 类型时，我们除了可以检查是否它们相等或不相等自己之外，如果它们不相等，我们还可以检查它们不相等是因为第一项小于第二项或者是第一项大于第二项。

默认情况下，`Rhs == Self`，因为我们总是想要比较同一类型的实例，而不是对不同类型的实例。这也自动保证了我们的实现是对称的和可传递的。

```

use std::cmp::Ordering;

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

// Rhs == Self == Point
implPartialOrd for Point {
    // impl automatically symmetric & transitive
    fn partial_cmp(&self, other: &Point) -> Option<Ordering> {
        Some(match self.x.cmp(&other.x) {
            Ordering::Equal => self.y.cmp(&other.y),
            ordering => ordering,
        })
    }
}

```

如果一个类型的所有成员都实现了 `PartialOrd`，那么它就可以被派生：

```

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

#[derive(PartialEq, PartialOrd)]
enum Stoplight {
    Red,
    Yellow,
    Green,
}

```

派生宏 `PartialOrd` 根据字典序（lexicographical）对它们的成员进行排序：

```
// generates PartialOrd impl which orders
// Points based on x member first and
// y member second because that's the order
// they appear in the source code
#[derive(PartialOrd, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

// generates DIFFERENT PartialOrd impl
// which orders Points based on y member
// first and x member second
#[derive(PartialOrd, PartialEq)]
struct Point {
    y: i32,
    x: i32,
}
```

Ord 是 **Eq** 和 **PartialOrd<Self>** 的一个 subtrait:

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;

    // provided default impls
    fn max(self, other: Self) -> Self;
    fn min(self, other: Self) -> Self;
    fn clamp(self, min: Self, max: Self) -> Self;
}
```

如果我们为一个类型实现了 **Ord**，在 **PartialOrd** 保证了非对称性和传递性之上，我们还能保证整体的非对称性，即对于任意给定的 **a**、**b**，**a < b**、**a == b** 或 **a > b** 中必有一个为真。从这个角度来讲，**Ord** 细化了 **Eq** 和 **PartialOrd**，因为它表示一个更严格的比较。如果一个类型实现了 **Ord**，我们就可以利用这个实现来实现 **PartialOrd**、**PartialEq** 和 **Eq**：

```

use std::cmp::Ordering;

// of course we can use the derive macros here
#[derive(Ord, PartialOrd, Eq, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

// note: as with PartialOrd, the Ord derive macro
// orders a type based on the lexicographical order
// of its members

// but here's the impls if we wrote them out by hand
implOrdfor Point {
    fn cmp(&self, other: &Self) -> Ordering {
        matchself.x.cmp(&other.x) {
            Ordering::Equal => self.y.cmp(&other.y),
            ordering => ordering,
        }
    }
}

implPartialOrdfor Point {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

implPartialEqfor Point {
    fn eq(&self, other: &Self) -> bool {
        self.cmp(other) == Ordering::Equal
    }
}

implEqfor Point {}

```

浮点型实现了 `PartialOrd` 但是没有实现 `Ord`，因为 `NaN < 0 == false` 和 `NaN >= 0 == false` 都为真。几乎所有的其他的 `PartialOrd` 类型都实现了 `Ord`，除非它们中包含有浮点型。

一旦一个类型实现了 `Ord`，我们就可以把它存储在 `BTreeMap` 和 `BTreeSet`，还可以在 `slice` 上使用 `sort()` 方法对其进行排序，这同样适用于其他可以解引用为 `slice` 的类型，比如数组、`Vec` 和 `VecDeque`。

```
use std::collections::BTreeSet;

// now our type can be stored
// in BTreeSets and BTreeMaps!
#[derive(Ord, PartialOrd, PartialEq, Eq)]
struct Point {
    x: i32,
    y: i32,
}

fn example_btreeset() {
    letmut points = BTreeSet::new();
    points.insert(Point { x: 0, y: 0 }); // ✅
}

// we can also .sort() Ord types in collections!
fn example_sort<T: Ord>(mut sortable: Vec<T>) -> Vec<T> {
    sortable.sort();
    sortable
}
```