【译】Rust标准库Trait指南（七）（完）

mp.weixin.qq.com/s/0iDdDUpKR0fDmgmLBjnjgg

> 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）

Iteration Traits

```
trait Iterator {
type Item;
fn next(&mutself) -> Option<Self::Item>;



// provided default impls
fn size_hint(&self) -> (usize, Option<usize>);
fn count(self) -> usize;
fn last(self) -> Option<Self::Item>;
fn advance_by(&mutself, n: usize) -> Result<(), usize>;
fn nth(&mutself, n: usize) -> Option<Self::Item>;
fn step_by(self, step: usize) -> StepBy<Self>;
fn chain<U>(
self,
        other: U
    ) -> Chain<Self, <U asIntoIterator>::IntoIter>
where
        U: IntoIterator<Item = Self::Item>;
fn zip<U>(self, other: U) -> Zip<Self, <U asIntoIterator>::IntoIter>
where
        U: IntoIterator;
fn map<B, F>(self, f: F) -> Map<Self, F>
where
        F: FnMut(Self::Item) -> B;
fn for_each<F>(self, f: F)
where
        F: FnMut(Self::Item);
fn filter<P>(self, predicate: P) -> Filter<Self, P>
where
        P: FnMut(&Self::Item) -> bool;
fn filter_map<B, F>(self, f: F) -> FilterMap<Self, F>
where
        F: FnMut(Self::Item) -> Option<B>;
fn enumerate(self) -> Enumerate<Self>;
fn peekable(self) -> Peekable<Self>;
fn skip_while<P>(self, predicate: P) -> SkipWhile<Self, P>
where
        P: FnMut(&Self::Item) -> bool;
fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
where
        P: FnMut(&Self::Item) -> bool;
fn map_while<B, P>(self, predicate: P) -> MapWhile<Self, P>
where
        P: FnMut(Self::Item) -> Option<B>;
fn skip(self, n: usize) -> Skip<Self>;
fn take(self, n: usize) -> Take<Self>;
fn scan<St, B, F>(self, initial_state: St, f: F) -> Scan<Self, St, F>
where
        F: FnMut(&mut St, Self::Item) -> Option<B>;
fn flat_map<U, F>(self, f: F) -> FlatMap<Self, U, F>
where
        F: FnMut(Self::Item) -> U,
        U: IntoIterator;
fn flatten(self) -> Flatten<Self>
where
        Self::Item: IntoIterator;
fn fuse(self) -> Fuse<Self>;
fn inspect<F>(self, f: F) -> Inspect<Self, F>
where
        F: FnMut(&Self::Item);
fn by_ref(&mutself) -> &mutSelf;
fn collect<B>(self) -> B
where
        B: FromIterator<Self::Item>;
fn partition<B, F>(self, f: F) -> (B, B)
where
        F: FnMut(&Self::Item) -> bool,
        B: Default + Extend<Self::Item>;
fn partition_in_place<'a, T, P>(self, predicate: P) -> usize
where
Self: DoubleEndedIterator<Item = &'amut T>,
        T: 'a,
        P: FnMut(&T) -> bool;
fn is_partitioned<P>(self, predicate: P) -> bool
where
        P: FnMut(Self::Item) -> bool;
fn try_fold<B, F, R>(&mutself, init: B, f: F) -> R
where
        F: FnMut(B, Self::Item) -> R,
        R: Try<Ok = B>;
fn try_for_each<F, R>(&mutself, f: F) -> R
where
        F: FnMut(Self::Item) -> R,
        R: Try<Ok = ()>;
fn fold<B, F>(self, init: B, f: F) -> B
where
        F: FnMut(B, Self::Item) -> B;
fn fold_first<F>(self, f: F) -> Option<Self::Item>
where
        F: FnMut(Self::Item, Self::Item) -> Self::Item;
fn all<F>(&mutself, f: F) -> bool
where
        F: FnMut(Self::Item) -> bool;
fn any<F>(&mutself, f: F) -> bool
where
        F: FnMut(Self::Item) -> bool;
fn find<P>(&mutself, predicate: P) -> Option<Self::Item>
where
        P: FnMut(&Self::Item) -> bool;
fn find_map<B, F>(&mutself, f: F) -> Option<B>
where
        F: FnMut(Self::Item) -> Option<B>;
fn try_find<F, R>(
        &mutself,
        f: F
    ) -> Result<Option<Self::Item>, <R as Try>::Error>
where
        F: FnMut(&Self::Item) -> R,
        R: Try<Ok = bool>;
fn position<P>(&mutself, predicate: P) -> Option<usize>
where
        P: FnMut(Self::Item) -> bool;
fn rposition<P>(&mutself, predicate: P) -> Option<usize>
where
Self: ExactSizeIterator + DoubleEndedIterator,
        P: FnMut(Self::Item) -> bool;
fn max(self) -> Option<Self::Item>
where
        Self::Item: Ord;
fn min(self) -> Option<Self::Item>
where
        Self::Item: Ord;
fn max_by_key<B, F>(self, f: F) -> Option<Self::Item>
where
        F: FnMut(&Self::Item) -> B,
        B: Ord;
fn max_by<F>(self, compare: F) -> Option<Self::Item>
where
        F: FnMut(&Self::Item, &Self::Item) -> Ordering;
fn min_by_key<B, F>(self, f: F) -> Option<Self::Item>
where
        F: FnMut(&Self::Item) -> B,
        B: Ord;
fn min_by<F>(self, compare: F) -> Option<Self::Item>
where
        F: FnMut(&Self::Item, &Self::Item) -> Ordering;
fn rev(self) -> Rev<Self>
where
Self: DoubleEndedIterator;
fn unzip<A, B, FromA, FromB>(self) -> (FromA, FromB)
where
Self: Iterator<Item = (A, B)>,
        FromA: Default + Extend<A>,
        FromB: Default + Extend<B>;
fn copied<'a, T>(self) -> Copied<Self>
where
Self: Iterator<Item = &'a T>,
        T: 'a + Copy;
fn cloned<'a, T>(self) -> Cloned<Self>
where
Self: Iterator<Item = &'a T>,
        T: 'a + Clone;
fn cycle(self) -> Cycle<Self>
where
Self: Clone;
fn sum<S>(self) -> S
where
        S: Sum<Self::Item>;
fn product<P>(self) -> P
where
        P: Product<Self::Item>;
fn cmp<I>(self, other: I) -> Ordering
where
        I: IntoIterator<Item = Self::Item>,
        Self::Item: Ord;
fn cmp_by<I, F>(self, other: I, cmp: F) -> Ordering
where
        F: FnMut(Self::Item, <I asIntoIterator>::Item) -> Ordering,
        I: IntoIterator;
fn partial_cmp<I>(self, other: I) -> Option<Ordering>
where
        I: IntoIterator,
        Self::Item: PartialOrd<<I asIntoIterator>::Item>;
```

```
fn partial_cmp_by<I, F>(
self,
        other: I,
        partial_cmp: F
    ) -> Option<Ordering>
where
        F: FnMut(Self::Item, <I asIntoIterator>::Item) -> Option<Ordering>,
        I: IntoIterator;
fn eq<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialEq<<I asIntoIterator>::Item>;
fn eq_by<I, F>(self, other: I, eq: F) -> bool
where
        F: FnMut(Self::Item, <I asIntoIterator>::Item) -> bool,
        I: IntoIterator;
fn ne<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialEq<<I asIntoIterator>::Item>;
fn lt<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialOrd<<I asIntoIterator>::Item>;
fn le<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialOrd<<I asIntoIterator>::Item>;
fn gt<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialOrd<<I asIntoIterator>::Item>;
fn ge<I>(self, other: I) -> bool
where
        I: IntoIterator,
        Self::Item: PartialOrd<<I asIntoIterator>::Item>;
fn is_sorted(self) -> bool
where
        Self::Item: PartialOrd<Self::Item>;
fn is_sorted_by<F>(self, compare: F) -> bool
where
        F: FnMut(&Self::Item, &Self::Item) -> Option<Ordering>;
fn is_sorted_by_key<F, K>(self, f: F) -> bool
where
        F: FnMut(Self::Item) -> K,
        K: PartialOrd<K>;
}
```

`Iterator<Item = T>` 类型可以被迭代并产生 `T` 类型。没有 `IteratorMut` trait。每个 `Iterator` 实现可以指定它返回的是不可变引用、可变引用还是拥有通过 `Item` 关联类型的值。

`Vec<T>` 方法	**返回**

`Vec<T>` 方法	返回
`.iter()`	`Iterator<Item = &T>`
`.iter_mut()`	`Iterator<Item = &mut T>`
`.into_iter()`	`Iterator<Item = T>`

大多数类型没有它们自己的迭代器，这对于初级 Rustaceans 来说，并不明显，但中级 Rustaceans 认为这是理所当然的。如果一个类型是可迭代的，我们几乎总是实现自定义的迭代器类型来迭代它，而不是让它自己迭代。

```
struct MyType {
    items: Vec<String>
}

impl MyType {
fn iter(&self) -> implIterator<Item = &String> {
        MyTypeIterator {
            index: 0,
            items: &self.items
        }
    }
}
```

```
struct MyTypeIterator<'a> {
    index: usize,
    items: &'aVec<String>
}
```

```
impl<'a> Iteratorfor MyTypeIterator<'a> {
type Item = &'aString;
fn next(&mutself) -> Option<Self::Item> {
ifself.index >= self.items.len() {
None
        } else {
let item = &self.items[self.index];
self.index += 1;
Some(item)
        }
    }
}
```

为了便于教学，上面的例子展示了如何从头开始实现一个迭代器，但在这种情况下，常用的解决方案是直接延用 `Vec` 的 `iter` 方法。

```
struct MyType {
    items: Vec<String>
}

impl MyType {
fn iter(&self) -> implIterator<Item = &String> {
self.items.iter()
    }
}
```

而且，这也是一个需要注意到的 generic blanket impl：

```
impl<I: Iterator + ?Sized> Iterator for &mut I;
```

一个迭代器的可变引用也是一个迭代器。知道这一点是有用的，因为它让我们能够使用 `self` 作为接收器（receiver）的迭代器方法，就像 `&mut self` 接收器一样。

举个例子，假定我们有一个函数，它处理一个数据超过三项的迭代器，但是函数的第一步是取出迭代器的前三项并在迭代完剩余项之前单独处理它们，下面是一个初学者可能会写出的函数实现：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
let first3: Vec<i32> = iter.take(3).collect();
for item in iter { // ❌ iter consumed in line above
// process remaining items
    }
}
```

这看起来有点让人头疼。 `take` 方法有一个 `self` 接收器，所以我们似乎不能在没有消耗整个迭代器的情况下调用它！下面是对上面代码的重构：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = vec![
        iter.next().unwrap(),
        iter.next().unwrap(),
        iter.next().unwrap(),
    ];
    for item in iter { // ✅
        // process remaining items
    }
}
```

这样是没问题的，但是实际中通常会这样重构：

```
fn example<I: Iterator<Item = i32>>(mut iter: I) {
    let first3: Vec<i32> = iter.by_ref().take(3).collect();
    for item in iter { // ✅
        // process remaining items
    }
}
```

这种写法不太常见，但不管怎样，现在我们知道了。

此外，对于什么类型可以或者不可以是迭代器，并没有规则或者约定。如果一个类型实现了 Iterator ，那么它就是一个迭代器。下面是标准库中一个新颖的例子：

```
use std::sync::mpsc::channel;
use std::thread;

fn paths_can_be_iterated(path: &Path) {
for part in path {
// iterate over parts of a path
    }
}



fn receivers_can_be_iterated() {
let (send, recv) = channel();



    thread::spawn(move || {
        send.send(1).unwrap();
        send.send(2).unwrap();
        send.send(3).unwrap();
    });



for received in recv {
// iterate over received values
    }
}
```

IntoIterator

```
trait IntoIterator
where
    <Self::IntoIter as Iterator>::Item == Self::Item,
{
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

正如其名， IntoIterator 类型可以转化为迭代器。当一个类型在一个 for-in 循环里被使用的时候，该类型的 into_iter 方法会被调用：

```
// vec = Vec<T>
for v in vec {} // v = T

// above line desugared
for v in vec.into_iter() {}
```

不仅 `Vec` 实现了 `IntoIterator` ，如果我们想在不可变引用或可变引用上迭代， `&Vec` 和 `&mut Vec` 同样也是如此。

```
// vec = Vec<T>
for v in &vec {} // v = &T

// above example desugared
for v in (&vec).into_iter() {}

// vec = Vec<T>
for v in &mut vec {} // v = &mut T

// above example desugared
for v in (&mut vec).into_iter() {}
```

FromIterator

```
trait FromIterator<A> {
fn from_iter<T>(iter: T) -> Self
where
        T: IntoIterator<Item = A>;
}
```

正如其名， `FromIterator` 类型可以从一个迭代器创建而来。 `FromIterator` 最常用于 `Iterator` 上的 `collect` 方法调用：

```
fn collect<B>(self) -> B
where
    B: FromIterator<Self::Item>;
```

下面是一个例子，搜集（collect）一个 `Iterator<Item = char>` 到 `String` ：

```
fn filter_letters(string: &str) -> String {
    string.chars().filter(|c| c.is_alphabetic()).collect()
}
```

标准库中所有的集合都实现了 `IntoIterator` 和 `FromIterator` ，从而使它们之间的转换更为简单：

```
use std::collections::{BTreeSet, HashMap, HashSet, LinkedList};

// String -> HashSet<char>
fn unique_chars(string: &str) -> HashSet<char> {
    string.chars().collect()
}

// Vec<T> -> BTreeSet<T>
fn ordered_unique_items<T: Ord>(vec: Vec<T>) -> BTreeSet<T> {
    vec.into_iter().collect()
}

// HashMap<K, V> -> LinkedList<(K, V)>
fn entry_list<K, V>(map: HashMap<K, V>) -> LinkedList<(K, V)> {
    map.into_iter().collect()
}

// and countless more possible examples
```

I/O Traits

```
trait Read {
fn read(&mutself, buf: &mut [u8]) -> Result<usize>;


// provided default impls
fn read_vectored(&mutself, bufs: &mut [IoSliceMut<'_>]) -> Result<usize>;
fn is_read_vectored(&self) -> bool;
unsafefn initializer(&self) -> Initializer;
fn read_to_end(&mutself, buf: &mutVec<u8>) -> Result<usize>;
fn read_to_string(&mutself, buf: &mutString) -> Result<usize>;
fn read_exact(&mutself, buf: &mut [u8]) -> Result<()>;
fn by_ref(&mutself) -> &mutSelf
where
Self: Sized;
fn bytes(self) -> Bytes<Self>
where
Self: Sized;
fn chain<R: Read>(self, next: R) -> Chain<Self, R>
where
Self: Sized;
fn take(self, limit: u64) -> Take<Self>
where
Self: Sized;
}



trait Write {
fn write(&mutself, buf: &[u8]) -> Result<usize>;
fn flush(&mutself) -> Result<()>;


// provided default impls
fn write_vectored(&mutself, bufs: &[IoSlice<'_>]) -> Result<usize>;
fn is_write_vectored(&self) -> bool;
fn write_all(&mutself, buf: &[u8]) -> Result<()>;
fn write_all_vectored(&mutself, bufs: &mut [IoSlice<'_>]) -> Result<()>;
fn write_fmt(&mutself, fmt: Arguments<'_>) -> Result<()>;
fn by_ref(&mutself) -> &mutSelf
where
Self: Sized;
}
```

值得关注的 generic blanket impls:

```
impl<R: Read + ?Sized> Read for &mut R;
impl<W: Write + ?Sized> Write for &mut W;
```

也就是说， `Read` 类型的任何可变引用也都是 `Read` ， `Write` 同理。知道这些是有用的，因为它允许我们使用任何带有 `self` 接收器的方法，就像它有一个 `&mut self` 接收器一样。我们已经在迭代器 trait 部分讲过了它是如何起作用的以及为什么很有用，所以这里不

再赘述。

这里我想指出的是， `&[u8]` 实现了 `Read` ， `Vec<u8>` 实现了 `Write` 。因此我们可以对我们的文件处理函数进行简单的单元测试，通过使用 `String` 转换为 `&[u8]` 以及从 `Vec<u8>` 转换为 `String` ：

```
use std::path::Path;
use std::fs::File;
use std::io::Read;
use std::io::Write;
use std::io;

// function we want to test
fn uppercase<R: Read, W: Write>(mut read: R, mut write: W) -
> Result<(), io::Error> {
letmut buffer = String::new();
    read.read_to_string(&mut buffer)?;
let uppercase = buffer.to_uppercase();
    write.write_all(uppercase.as_bytes())?;
    write.flush()?;
Ok(())
}



// in actual program we'd pass Files
fn example(in_path: &Path, out_path: &Path) -> Result<(), io::Error> {
let in_file = File::open(in_path)?;
let out_file = File::open(out_path)?;
    uppercase(in_file, out_file)
}

        // however in unit tests we can use Strings!
#[test] // ✅
fn example_test() {
    let in_file: String = "i am screaming".into();
    let mut out_file: Vec<u8> = Vec::new();
    uppercase(in_file.as_bytes(), &mut out_file).unwrap();
    let out_result = String::from_utf8(out_file).unwrap();
    assert_eq!(out_result, "I AM SCREAMING");
}
```

总结

我们一起学到了很多！事实上是太多了。这是我们现在的样子：

Hi! My name is Jason
Ah! Nice to meet you, Jason
Jason
FRIENDS
FAMILY
Rust standard library traits
Say Jarvis, wanna hear the difference between AsRef<T> and Borrow<T>?
Rust碎碎念

Rust碎碎念
长按二维码识别　一键关注新动向
Rust碎碎念