

【译】Rust标准库Trait指南（五）

 mp.weixin.qq.com/s/TKDNB4zzAZEs_RFTOp2nw

原创 Rust碎碎念 Rust碎碎念 7月12日

收录于话题

#Rust 14 个内容

#Rust翻译文章集合 60 个内容

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 

算术 Trait (Arithmetic Traits)

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
Add	算术	+	相加
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

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
<code>Div</code>	算术	<code>/</code>	除
<code>DivAssign</code>	算术	<code>/=</code>	除并赋值
<code>Mul</code>	算术	<code>*</code>	乘
<code>MulAssign</code>	算术	<code>*=</code>	乘并赋值
<code>Neg</code>	算术	<code>-</code>	一元求反
<code>Not</code>	算术	<code>!</code>	一元逻辑求反
<code>Rem</code>	算术	<code>%</code>	求余
<code>RemAssign</code>	算术	<code>%=</code>	求余并赋值
<code>Shl</code>	算术	<code><<</code>	左移
<code>ShlAssign</code>	算术	<code><<=</code>	左移并赋值
<code>Shr</code>	算术	<code>>></code>	右移
<code>ShrAssign</code>	算术	<code>>>=</code>	右移并赋值
<code>Sub</code>	算术	<code>-</code>	减
<code>SubAssign</code>	算术	<code>-=</code>	减并赋值

我们没有必要把所有的算术操作符都仔细看一遍，毕竟它们中大多数都只作用于数值类型。我们将会讨论 `Add` 和 `AddAssign`，因为 `+` 操作符经常被重载用来完成其他事情，比如往集合里添加一项，或者进行拼接操作，这样我们就可以从最有趣的地方入手而不会重复。

Add & AddAssign

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

`Add<Rhs, Output = T>` 类型可以被加到 `Rhs` 类型上并产生一个 `T` 作为输出。

例如，在 `Point` 上实现 `Add<Point, Output = Point>`：

```

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

impl Add for Point {
    type Output = Point;
    fn add(self, rhs: Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = p1 + p2;
    assert_eq!(p3.x, p1.x + p2.x); // ✓
    assert_eq!(p3.y, p1.y + p2.y); // ✓
}

```

但是，如果我们只有 `Point` 的引用，那该怎么办呢？我们还能把它们相加么？让我们试试：

```

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = &p1 + &p2; // ✗
}

```

显然不可以，编译器抛出下面的提示：

```

error[E0369]: cannot add `&Point` to `&Point`
--> src/main.rs:50:25
50 |         let p3: Point = &p1 + &p2;
    |                        --- ^ --- &Point
    |                        |
    |                        &Point
    = note: an implementation of `std::ops::Add` might be missing for `&Point`

```

在 Rust 的类型系统中，对于某个类型 `T`，`T`、`&T`、`&mut T` 都会被视作是完全不同的类型，这意味着我们必须分别为它们提供 trait 的实现。让我们为 `&Point` 实现 `Add`：

```
impl Add for &Point {
    type Output = Point;
    fn add(self, rhs: &Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let p3 = &p1 + &p2; // ✓
    assert_eq!(p3.x, p1.x + p2.x); // ✓
    assert_eq!(p3.y, p1.y + p2.y); // ✓
}
```

尽管如此，但是仍然感觉有些地方不太对。我们针对 `Point` 和 `&Point` 实现了两份 `Add`，它们恰好目前还做了相同的事情，但是我们不能保证将来也是如此。例如，假设我们决定，当我们把两个 `Point` 相加时，我们想要创建一个包含这两个 `Point` 的 `Line` 类型而不是创建一个新的 `Point`，那么我们会把 `Add` 的实现更新：

```

use std::ops::Add;

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

#[derive(Copy, Clone)]
struct Line {
    start: Point,
    end: Point,
}

// we updated this impl
impl Add for Point {
    type Output = Line;
    fn add(self, rhs: Point) -> Line {
        Line {
            start: self,
            end: rhs,
        }
    }
}

// but forgot to update this impl, uh oh!
impl Add for &Point {
    type Output = Point;
    fn add(self, rhs: &Point) -> Point {
        Point {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = p1 + p2; // ✅

    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = &p1 + &p2; // ❌ expected Line, found Point
}

```

我们当前针对 `&Point` 的 `Add` 实现就产生了一个不必要的维护负担，我们希望这个实现能够自动匹配 `Point` 的实现而无需我们每次在修改 `Point` 的实现时都手动维护更新。我们想要保持我们的代码尽可能地 DRY（Don't Repeat Yourself，不要重复自己）。幸运的是这是可以实现的：

```
// updated, DRY impl
impl Add for &Point {
    type Output = <Point as Add>::Output;
    fn add(self, rhs: &Point) -> Self::Output {
        Point::add(*self, *rhs)
    }
}
```

```
fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    let line: Line = p1 + p2; // ✓
```

```
let p1 = Point { x: 1, y: 2 };
let p2 = Point { x: 3, y: 4 };
let line: Line = &p1 + &p2; // ✓
}
```

AddAssign<Rhs> 类型能够让我们和 **Rhs** 类型相加并赋值。该 **trait** 声明如下：

```
trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}
```

以 **Point** 和 **&Point** 为例：

```
use std::ops::AddAssign;

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

impl AddAssign for Point {
    fn add_assign(&mutself, rhs: Point) {
        self.x += rhs.x;
        self.y += rhs.y;
    }
}

impl AddAssign<&Point> for Point {
    fn add_assign(&mutself, rhs: &Point) {
        Point::add_assign(self, *rhs);
    }
}

fn main() {
    letmut p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 3, y: 4 };
    p1 += &p2;
    p1 += p2;
    assert!(p1.x == 7 && p1.y == 10);
}
```

闭包 Trait（Closure Traits）



Trait(s)	分类（Category）	操作符（Operator(s)）	描述（Description）
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

FnOnce, FnMut, & Fn

```
trait FnOnce<Args> {  
    type Output;  
    fn call_once(self, args: Args) -> Self::Output;  
}  
  
trait FnMut<Args>: FnOnce<Args> {  
    fn call_mut(&mutself, args: Args) -> Self::Output;  
}  
  
trait Fn<Args>: FnMut<Args> {  
    fn call(&self, args: Args) -> Self::Output;  
}
```

虽然存在这些 trait，但是在 stable 的 Rust 中，我们无法为自己的类型实现这些 trait。我们能够创建的唯一能够实现这些 trait 的类型就是闭包。闭包根据其从环境中所捕获的内容来决定它到底是实现 `FnOnce`、`FnMut` 还是 `Fn`。

`FnOnce` 闭包只能被调用一次，因为它会在执行过程中消耗掉某些值：

```
fn main() {  
    let range = 0..10;  
    let get_range_count = || range.count();  
    assert_eq!(get_range_count(), 10); //   
    get_range_count(); //   
}
```

迭代器上的 `.count()` 方法会消耗迭代器，因此它只能被调用一次。因此，我们的闭包也只能调用一次。这也是为什么我们在尝试调用第二次的时候会得到下面的错误：


```

error[E0382]: use of moved value: `get_range_count`
--> src/main.rs:5:5
|
4 |     assert_eq!(get_range_count(), 10);
|                  ^^^^^^^^^^^^^^^^^ `get_range_count` moved due to this call
5 |     get_range_count();
|     ^^^^^^^^^^^^^^^^^ value used here after move
|
note: closure cannot be invoked more than once because it moves the variable `range`
--> src/main.rs:3:30
|
3 |     let get_range_count = || range.count();
|                               ^^^^^
note: this value implements `FnOnce`, which causes it to be moved when called
--> src/main.rs:4:16
|
4 |     assert_eq!(get_range_count(), 10);
|                  ^^^^^^^^^^^^^^^^^

```

FnMut 闭包可以被多次调用，并且可以修改它从环境中捕获到的变量。我们可以说 **FnMut** 有副作用或者是有状态的（stateful）。下面是一个闭包的示例，通过从迭代器中追踪它见到的最小值来过滤所有非升序的值。


```

fn main() {
    let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];
    letmut min = i32::MIN;
    let ascending = nums.into_iter().filter(|&n| {
        if n <= min {
            false
        } else {
            min = n;
            true
        }
    }).collect::<Vec<_>>();
    assert_eq!(vec![0, 4, 8, 10, 15, 18], ascending); // ✓
}

```

FnOnce 会获取它的参数的所有权并且只能被调用一次，但是 **FnMut** 仅要求获取参数的可变引用并且可以被多次调用，从这一点上来讲，**FnMut** 细化了 **FnOnce**。**FnMut** 可以用于任何可以使用 **FnOnce** 的地方。


Fn 闭包也可以被调用多次，但是它不能修改从环境中捕获的变量。我们可以说，**Fn** 闭包没有副作用或者无状态的（stateless）。下面是一个示例，从一个迭代器中过滤出所有小于某个栈上变量的数字，该变量是它是环境中捕获到的：


```
fn main() {
    let nums = vec![0, 4, 2, 8, 10, 7, 15, 18, 13];
    let min = 9;
    let greater_than_9 = nums.into_iter().filter(|&n| n > min).collect::<Vec<_>>();
    assert_eq!(vec![10, 15, 18, 13], greater_than_9); // 
}
```

FnMut 要求可变引用并且可以被多次调用，**Fn** 只要求不可变引用并可以被多次调用，从这一点来讲，**Fn** 细化了 **FnMut**。**Fn** 可以被用于任何可以使用 **FnMut** 的地方，当然也包括可以使用 **FnOnce** 的地方。


如果一个闭包不从环境中捕获任何变量，从技术角度来讲它算不上是闭包，而只是一个被匿名声明的内联函数，并且可以作为一个普通函数指针（即 **Fn**）被使用和传递，这包括可以使用 **FnMut** 和 **FnOnce** 的地方。

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn main() {
    letmut fn_ptr: fn(i32) -> i32 = add_one;
    assert_eq!(fn_ptr(1), 2); // 

    // capture-less closure cast to fn pointer
    fn_ptr = |x| x + 1; // same as add_one
    assert_eq!(fn_ptr(1), 2); // 
}
```

下面是一个传递普通函数指针而不是闭包的示例：

```
fn main() {
    let nums = vec![-1, 1, -2, 2, -3, 3];
    let absolutes: Vec<i32> = nums.into_iter().map(i32::abs).collect();
    assert_eq!(vec![1, 1, 2, 2, 3, 3], absolutes); // 
}
```

其他 Trait（Other Traits）

Trait(s)	分类（Category）	操作符（Operator(s)）	描述（Description）
Deref	其他	*	不可变解引用
DerefMut	其他	*	可变的解引用

Trait(s)	分类 (Category)	操作符 (Operator(s))	描述 (Description)
<code>Drop</code>	其他	-	类型析构
<code>Index</code>	其他	<code>[]</code>	不可变索引
<code>IndexMut</code>	其他	<code>[]</code>	可变索引
<code>RangeBounds</code>	其他	<code>..</code>	区间

```

trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

    trait DerefMut: Deref {
        fn deref_mut(&mut self) -> &mut Self::Target;
    }

```

`Deref<Target = T>` 类型可以使用 `*` 操作符解引用为 `T` 类型。这在像 `Box` 和 `Rc` 这样的智能指针类型中有很明显的用例。尽管如此，但是我们在 Rust 代码中很少见到这种显式的解引用操作，这是因为 Rust 有一个被称为 **解引用强制转换 (deref coercion)** 的特性。

当类型被作为函数参数传递、从函数返回或者作为方法调用的一部分时，Rust 会自动对这些类型进行解引用。这也解释了为什么我们可以在一个期望 `&str` 和 `&[T]` 的函数中可以传入 `&String` 和 `&Vec<T>`，因为 `String` 实现了 `Deref<Target = str>` 并且 `Vec<T>` 实现了 `Deref<Target = [T]>`。

`Deref` 和 `DerefMut` 应该仅被实现于智能指针类型。人们误用和滥用这些 trait 的最常见的方式是，试图把 OOP（面向对象程序设计）风格的数据继承塞进 Rust 中。这样是行不通的。Rust 不是 OOP。让我们进行一些测试，来看看它是在哪里、怎么样以及为什么行不通。让我们从下面的例子开始：

```
use std::ops::Deref;

struct Human {
    health_points: u32,
}

enum Weapon {
    Spear,
    Axe,
    Sword,
}

// a Soldier is just a Human with a Weapon
struct Soldier {
    human: Human,
    weapon: Weapon,
}

impl Deref for Soldier {
    type Target = Human;
    fn deref(&self) -> &Human {
        &self.human
    }
}

enum Mount {
    Horse,
    Donkey,
    Cow,
}

// a Knight is just a Soldier with a Mount
struct Knight {
    soldier: Soldier,
    mount: Mount,
}

impl Deref for Knight {
    type Target = Soldier;
    fn deref(&self) -> &Soldier {
        &self.soldier
    }
}

enum Spell {
    MagicMissile,
    FireBolt,
    ThornWhip,
}

// a Mage is just a Human who can cast Spells
struct Mage {
    human: Human,
    spells: Vec<Spell>,
}
```

```

impl Deref for Mage {
    type Target = Human;
    fn deref(&self) -> &Human {
        &self.human
    }
}

enum Staff {
    Wooden,
    Metallic,
    Plastic,
}

// a Wizard is just a Mage with a Staff
struct Wizard {
    mage: Mage,
    staff: Staff,
}

impl Deref for Wizard {
    type Target = Mage;
    fn deref(&self) -> &Mage {
        &self.mage
    }
}

fn borrows_human(human: &Human) {}
fn borrows_soldier(soldier: &Soldier) {}
fn borrows_knight(knight: &Knight) {}
fn borrows_mage(mage: &Mage) {}
fn borrows_wizard(wizard: &Wizard) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types can be used as Humans
    borrows_human(&human);
    borrows_human(&soldier);
    borrows_human(&knight);
    borrows_human(&mage);
    borrows_human(&wizard);
    // Knights can be used as Soldiers
    borrows_soldier(&soldier);
    borrows_soldier(&knight);
    // Wizards can be used as Mages
    borrows_mage(&mage);
    borrows_mage(&wizard);
    // Knights & Wizards passed as themselves
    borrows_knight(&knight);
    borrows_wizard(&wizard);
}

```

乍看之下，上面的代码似乎还不错！但是，仔细观察之后它就没这么好了。首先，解引用强制转换仅作用于引用，因此，当我们想要传递所有权的时候它是行不通的：

```
fn takes_human(human: Human) {}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types CANNOT be used as Humans
    takes_human(human);
    takes_human(soldier); // ✗
    takes_human(knight); // ✗
    takes_human(mage); // ✗
    takes_human(wizard); // ✗
}
```

此外，解引用强制转换在泛型上下文中是无法工作的。假定我们仅在 `humans` 上实现某个 `trait`：

```
trait Rest {
    fn rest(&self);
}

impl Rest for Human {
    fn rest(&self) {}
}

fn take_rest<T: Rest>(rester: &T) {
    rester.rest()
}

fn example(human: Human, soldier: Soldier, knight: Knight, mage: Mage, wizard: Wizard) {
    // all types CANNOT be used as Rest types, only Human
    take_rest(&human);
    take_rest(&soldier); // ✗
    take_rest(&knight); // ✗
    take_rest(&mage); // ✗
    take_rest(&wizard); // ✗
}
```

而且，尽管解引用强制转换在很多场景都可以使用，但它不是万能的。它无法作用于操作数，尽管操作符只是方法调用的语法糖。假定，我们想要 `Mage`（魔术师）通过 `+=` 操作符学会 `Spell`（拼写）：

```
impl DerefMut for Wizard {  
    fn deref_mut(&mutself) -> &mut Mage {  
        &mutself.mage  
    }  
}
```

```
impl AddAssign<Spell> for Mage {  
    fn add_assign(&mutself, spell: Spell) {  
        self.spells.push(spell);  
    }  
}
```

```
fn example(mut mage: Mage, mut wizard: Wizard, spell: Spell) {  
    mage += spell;  
    wizard += spell; // ❌ wizard not coerced to mage here  
    wizard.add_assign(spell); // oof, we have to call it like this 🤖  
}
```

在具有 OOP 风格的数据继承的编程语言中，一个方法中的 `self` 的值总是等于调用这个方法类型，但是在 Rust 中，`self` 的值永远等于实现这个方法类型：

```

struct Human {
    profession: &'staticstr,
    health_points: u32,
}

impl Human {
    // self will always be a Human here, even if we call it on a Soldier
    fn state_profession(&self) {
        println!("I'm a {}!", self.profession);
    }
}

struct Soldier {
    profession: &'staticstr,
    human: Human,
    weapon: Weapon,
}

fn example(soldier: &Soldier) {
    assert_eq!("servant", soldier.human.profession);
    assert_eq!("spearman", soldier.profession);
    soldier.human.state_profession(); // prints "I'm a servant!"
    soldier.state_profession(); // still prints "I'm a servant!" 🤖
}

```

当在一个新类型上实现 **Deref** 或 **DerefMut** 时，上面的陷阱令人震惊。假定我们想要创建一个 **SortedVec** 类型，它就是一个 **Vec** 只不过是有序的。下面是我们可能的实现方式：

```

struct SortedVec<T: Ord>(Vec<T>);

impl<T: Ord> SortedVec<T> {
    fn new(mut vec: Vec<T>) -> Self {
        vec.sort();
        SortedVec(vec)
    }
    fn push(&mutself, t: T) {
        self.0.push(t);
        self.0.sort();
    }
}

```

显然，这里我们不能实现 **DerefMut<Target = Vec<T>>**，否则任何使用 **SortedVec** 的人都能轻易打破已排好的顺序。但是，实现 **Deref<Target = Vec<T>>** 就一定安全么？试试找出下面程序中的 bug：


```

use std::ops::Deref;

struct SortedVec<T: Ord>(Vec<T>);

impl<T: Ord> SortedVec<T> {
    fn new(mut vec: Vec<T>) -> Self {
        vec.sort();
        SortedVec(vec)
    }
    fn push(&mutself, t: T) {
        self.0.push(t);
        self.0.sort();
    }
}

impl<T: Ord> Deref for SortedVec<T> {
    type Target = Vec<T>;
    fn deref(&self) -> &Vec<T> {
        &self.0
    }
}

fn main() {
    let sorted = SortedVec::new(vec![2, 8, 6, 3]);
    sorted.push(1);
    let sortedClone = sorted.clone();
    sortedClone.push(4);
}

```

我们未曾给 `SortedVec` 实现 `Clone`，所以当我们调用 `.clone()` 方法时，编译器使用解引用强制转换把它解析为 `Vec` 上的方法调用，所以它会返回一个 `Vec` 而不是一个 `SortedVec`！

```

fn main() {
    let sorted: SortedVec<i32> = SortedVec::new(vec![2, 8, 6, 3]);
    sorted.push(1); // still sorted

    // calling clone on SortedVec actually returns a Vec 🤖
    let sortedClone: Vec<i32> = sorted.clone();
    sortedClone.push(4); // sortedClone no longer sorted 🧟
}

```

不管怎样，上面的限制、约束或者陷阱都不是 Rust 的错，因为 Rust 从来都没有被设计成一门 OO（面向对象）的语言或者把支持 OOP（面向对象程序设计）模式放在首位。

本节的要点在于不要试图在 `Deref` 和 `DerefMut` 的实现耍小聪明。它们仅仅适用于智能指针类型，目前只能在标准库中实现，因为智能指针类型目前需要 `unstable` 的特性和编译器的魔法才能工作。如果我们想要类似于 `Deref` 和 `DerefMut` 的功能和行为，我们可以去了解一下后面会提到的 `AsRef` 和 `AsMut`。

Index & IndexMut

```
trait Index<Idx: ?Sized> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}
```

```
trait IndexMut<Idx>: Index<Idx> where Idx: ?Sized {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

我们可以将 `[]` 索引到带有 `T` 值的 `Index<T, Output = U>` 类型，索引操作将返回 `&U` 值。为了语法方便，编译器会自动在索引操作返回值的前面插入一个解引用操作符 `*`：

```
fn main() {
    // Vec<i32> impls Index<usize, Output = i32> so
    // indexing Vec<i32> should produce &i32s and yet...
    let vec = vec![1, 2, 3, 4, 5];
    let num_ref: &i32 = vec[0]; // ❌ expected &i32 found i32

    // above line actually desugars to
    let num_ref: &i32 = *vec[0]; // ❌ expected &i32 found i32

    // both of these alternatives work
    let num: i32 = vec[0]; // ✅
    let num_ref = &vec[0]; // ✅
}
```

为了展示我们自己如何实现 `Index`，下面是一个有趣的示例，这个例子展示了我们如何使用一个新类型和 `Index` trait 在 `Vec` 上实现环绕索引和非负索引：


```


use std::ops::Index;


struct WrappingIndex<T>(Vec<T>);


impl<T> Index<usize> for WrappingIndex<T> {
    type Output = T;
    fn index(&self, index: usize) -> &T {
        &self.0[index % self.0.len()]
    }
}

impl<T> Index<i128> for WrappingIndex<T> {
    type Output = T;
    fn index(&self, index: i128) -> &T {
        let self_len = self.0.len() as i128;
        let idx = (((index % self_len) + self_len) % self_len) as usize;
        &self.0[idx]
    }
}

#[test]// 
fn indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[0_usize]);
    assert_eq!(2, wrapping_vec[1_usize]);
    assert_eq!(3, wrapping_vec[2_usize]);
}

#[test]// 
fn wrapping_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[3_usize]);
    assert_eq!(2, wrapping_vec[4_usize]);
    assert_eq!(3, wrapping_vec[5_usize]);
}

#[test]// 
fn neg_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[-3_i128]);
    assert_eq!(2, wrapping_vec[-2_i128]);
    assert_eq!(3, wrapping_vec[-1_i128]);
}

#[test]// 
fn wrapping_neg_indexes() {
    let wrapping_vec = WrappingIndex(vec![1, 2, 3]);
    assert_eq!(1, wrapping_vec[-6_i128]);
}

```

```
assert_eq!(2, wrapping_vec[-5_i128]);
assert_eq!(3, wrapping_vec[-4_i128]);
}
```

这里没有要求 `Idx` 类型是数值类型或者是一个 `Range`，它也可以是一个枚举！下面是一个使用篮球位置在一支球队里检索球员的例子：

```
use std::ops::Index;

enum BasketballPosition {
    PointGuard,
    ShootingGuard,
    Center,
    PowerForward,
    SmallForward,
}

struct BasketballPlayer {
    name: &'staticstr,
    position: BasketballPosition,
}

struct BasketballTeam {
    point_guard: BasketballPlayer,
    shooting_guard: BasketballPlayer,
    center: BasketballPlayer,
    power_forward: BasketballPlayer,
    small_forward: BasketballPlayer,
}

impl Index<BasketballPosition> for BasketballTeam {
    type Output = BasketballPlayer;
    fn index(&self, position: BasketballPosition) -> &BasketballPlayer {
        match position {
            BasketballPosition::PointGuard => &self.point_guard,
            BasketballPosition::ShootingGuard => &self.shooting_guard,
            BasketballPosition::Center => &self.center,
            BasketballPosition::PowerForward => &self.power_forward,
            BasketballPosition::SmallForward => &self.small_forward,
        }
    }
}
```

Drop

```
trait Drop {
    fn drop(&mut self);
}
```

如果一个类型实现了 `Drop`，那么 `drop` 将会在该类型离开作用域但是销毁之前被调用。我们很少需要去为我们的类型实现它，但是如果一个类型中持有某些外部资源，这些资源需要在类型销毁时被清理，这种情况下就会用到了。

标准库中有一个 `BufWriter` 类型让我们能够把写入的数据缓冲到 `Write` 类型中。但是，如果 `BufWriter` 在它里面的内容被刷入到底层的 `Write` 类型之前就被销毁了，该怎么办呢？幸运的是那是不可能的！`BufWriter` 实现了 `Drop` trait，因此，无论什么它什么时候离开作用域，`flush` 总会被调用！

```
impl<W: Write> Drop for BufWriter<W> {  
    fn drop(&mut self) {  
        self.flush_buf();  
    }  
}
```

此外，Rust 中的 `Mutex` 没有 `unlock()` 方法，因为它们不需要！在 `Mutex` 上调用 `lock()` 会返回一个 `MutexGuard`，当 `MutexGuard` 离开作用域时，它会自动解锁（`unlock`）`Mutex`，这要归功于它的 `Drop` 实现：

```
impl<T: ?Sized> Drop for MutexGuard<'_, T> {  
    fn drop(&mut self) {  
        unsafe {  
            self.lock.inner.raw_unlock();  
        }  
    }  
}
```

一般而言，如果你正在实现对某类资源的抽象，这类资源需要在使用后被清理，那就是时候充分利用 `Drop` trait 了。

