

【译】Rust标准库Trait指南（六）

mp.weixin.qq.com/s/qtrlrws3Uz24LOERMpnggw

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 

转换 Traits (Conversion Traits)

From & Into

```
trait From<T> {  
    fn from(T) -> Self;  
}
```

From<T> 类型允许我们把 **T** 转换为 **Self**。

```
trait Into<T> {  
    fn into(self) -> T;  
}
```

Into<T> 类型允许我们把 **Self** 转换为 **T**。它们就像是一个硬币的两面。我们只能为自己的类型实现 **From<T>**，因为 **Into<T>** 的实现会通过 generic blanket impl 自动提供：

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

这两个 trait 之所以存在，是因为它能够让我们以稍微不同的方式来进行 trait 约束 (bound)：

```
fn function<T>(t: T)
where
    // these bounds are equivalent
    T: From<i32>,
    i32: Into<T>
{
    // these examples are equivalent
    let example: T = T::from(0);
    let example: T = 0.into();
}
```

没有规则强制要求什么时候使用前者或后者，所以在每种情景下采用最合理的方式就可以了。现在让我们来看一个例子：

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

implFrom<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Self {
        Point { x, y }
    }
}

implFrom<[i32; 2]> for Point {
    fn from([x, y]: [i32; 2]) -> Self {
        Point { x, y }
    }
}

fn example() {
    // 使用 From
    let origin = Point::from((0, 0));
    let origin = Point::from([0, 0]);

    // 使用 Into
    let origin: Point = (0, 0).into();
    let origin: Point = [0, 0].into();
}
```

这个实现不是对称的，因此，如果我们想要把 `Point` 转为 tuple 和 array，我们必须显式地添加下面的内容：

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

implFrom<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Self {
        Point { x, y }
    }
}

implFrom<Point> for (i32, i32) {
    fn from(Point { x, y }: Point) -> Self {
        (x, y)
    }
}

implFrom<[i32; 2]> for Point {
    fn from([x, y]: [i32; 2]) -> Self {
        Point { x, y }
    }
}

implFrom<Point> for [i32; 2] {
    fn from(Point { x, y }: Point) -> Self {
        [x, y]
    }
}

fn example() {
    // 从 (i32, i32) 到 Point
    let point = Point::from((0, 0));
    let point: Point = (0, 0).into();

    // 从 Point 到 (i32, i32)
    let tuple = <(i32, i32)>::from(point);
    let tuple: (i32, i32) = point.into();

    // 从 [i32; 2] 到 Point
    let point = Point::from([0, 0]);
    let point: Point = [0, 0].into();

    // 从 Point 到 [i32; 2]
    let array = <[i32; 2]>::from(point);
    let array: [i32; 2] = point.into();
}
```

`From<T>` 的一个常见用法是精简模板代码。假定我们想要在程序中添加一个 `Triangle` 类型，它里面包含三个 `Point`，下面是我们可以构造它的方式：

```

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

impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }
}

impl From<(i32, i32)> for Point {
    fn from((x, y): (i32, i32)) -> Point {
        Point { x, y }
    }
}

struct Triangle {
    p1: Point,
    p2: Point,
    p3: Point,
}

impl Triangle {
    fn new(p1: Point, p2: Point, p3: Point) -> Triangle {
        Triangle { p1, p2, p3 }
    }
}

impl<P> From<[P; 3]> for Triangle
where
    P: Into<Point>
{
    fn from([p1, p2, p3]: [P; 3]) -> Triangle {
        Triangle {
            p1: p1.into(),
            p2: p2.into(),
            p3: p3.into(),
        }
    }
}

fn example() {
    // 手动构造
    let triangle = Triangle {
        p1: Point {
            x: 0,
            y: 0,
        },
        p2: Point {
            x: 1,

```

```

        y: 1,
    },
    p3: Point {
        x: 2,
        y: 2,
    },
};

```

```

// 使用 Point::new
let triangle = Triangle {
    p1: Point::new(0, 0),
    p2: Point::new(1, 1),
    p3: Point::new(2, 2),
};

```

```

// 使用 From<(i32, i32)> for Point
let triangle = Triangle {
    p1: (0, 0).into(),
    p2: (1, 1).into(),
    p3: (2, 2).into(),
};

```

```

// 使用 Triangle::new + From<(i32, i32)> for Point
let triangle = Triangle::new(
    (0, 0).into(),
    (1, 1).into(),
    (2, 2).into(),
);

```

```

// 使用 From<[Into<Point>; 3]> for Triangle
let triangle: Triangle = [
    (0, 0),
    (1, 1),
    (2, 2),
].into();
}

```

关于你应该什么时候，以什么方式、什么理由来为我们的类型实现 `From<T>`，并没有强制规定，这取决于你对具体情况判断。

`Into<T>` 一个常见的用途是，使得需要拥有值的函数具有通用性，而不必关心它们是拥有值还是借用值。

```

struct Person {
    name: String,
}

impl Person {
    // 接受:
    // - String
    fn new1(name: String) -> Person {
        Person { name }
    }

    // 接受:
    // - String
    // - &String
    // - &str
    // - Box<str>
    // - Cow<'_, str>
    // - char
    // 因为上面所有的类型都可以转换为 String
    fn new2<N: Into<String>>(name: N) -> Person {
        Person { name: name.into() }
    }
}

```

错误处理（Error Handling）

讨论错误处理和 **Error** trait 的最好时机应该是紧跟在 **Display**、**Debug**、**Any**、**From** 之后，但是在 **TryFrom** 之前，这也是为什么把错误处理部分尴尬地嵌入在转换 trait 之间。

Error

```

trait Error: Debug + Display {
    // 提供默认实现
    fn source(&self) -> Option<&(dyn Error + 'static)>;
    fn backtrace(&self) -> Option<&Backtrace>;
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&dyn Error>;
}

```

在 Rust 中，错误（error）是被返回（return）的，而不是被抛出（throw）的，让我们看个例子。

因为整数除以 0 会 panic，如果我们想要让我们的程序更为安全，我们可以实现一个 **safe_div** 函数，它会返回一个 **Result**，就像下面这样：


```


use std::fmt;
use std::error;

#[derive(Debug, PartialEq)]
struct DivByZero;

impl fmt::Display for DivByZero {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "division by zero error")
    }
}

impl error::Error for DivByZero {}

fn safe_div(numerator: i32, denominator: i32) -> Result<i32, DivByZero> {
    if denominator == 0 {
        return Err(DivByZero);
    }
    Ok(numerator / denominator)
}

#[test]// 
fn test_safe_div() {
    assert_eq!(safe_div(8, 2), Ok(4));
    assert_eq!(safe_div(5, 0), Err(DivByZero));
}

```

因为错误是被返回而不是被抛出，所以这些错误必须被显式地处理，如果当前函数无法处理错误，该函数应该把错误传递给自己的调用者。传递错误的最常用方式是使用 **?** 操作符，它是现在已经弃用的 **try!** 宏的语法糖：

```

macro_rules! try {
    ($expr:expr) => {
        match $expr {
            // if Ok just unwrap the value
            Ok(val) => val,
            // if Err map the err value using From and return
            Err(err) => {
                return Err(From::from(err));
            }
        }
    };
}

```

如果我们想要写一个函数，该函数读取文件内容到 **String** 里，我们可以像这样写：

```

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

fn read_file_to_string(path: &Path) -> Result<String, io::Error> {
    let mut file = File::open(path)?; // ⬆ io::Error
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // ⬆ io::Error
    Ok(contents)
}

```

假定我们当前正在读取的文件内容是一串数字，并且我们想要把这些数字求和，我们可能会把函数更新成这样：

```

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

fn sum_file(path: &Path) -> Result<i32, /*这里放置什么? */> {
    let mut file = File::open(path)?; // ⬆ io::Error
    let mut contents = String::new();
    file.read_to_string(&mut contents)?; // ⬆ io::Error
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?; // ⬆ ParseIntError
    }
    Ok(sum)
}

```

但是，现在我们的 `Result` 里的错误类型应该是什么？它要么返回一个 `io::Error`，要么返回一个 `ParseIntError`。我们尝试寻找第三种方式来解决这个问题，以最快最乱的方式开始，以最健壮的方式结束。

第一种方式就是，识别出所有实现了 `Error` 和 `Display` 的类型，这样我们把所有的错误映射（map）到 `String` 类型并把 `String` 作为我们的错误类型：

```

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

fn sum_file(path: &Path) -> Result<i32, String> {
    let mut file = File::open(path)
        .map_err(|e| e.to_string())?; // ↑ io::Error -> String
    let mut contents = String::new();
    file.read_to_string(&mut contents)
        .map_err(|e| e.to_string())?; // ↑ io::Error -> String
    let mut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()
            .map_err(|e| e.to_string())?; // ↑ ParseIntError -> String
    }
    Ok(sum)
}

```

但是，这种方式的缺点在于，我们会丢弃所有的错误类型信息，从而导致调用者在处理错误时十分困难。

另外一个不太明显的优点则是，我们可以定制字符串来提供更多的特定上下文信息。例如，`ParseIntError` 通常会变成字符串 `"invalid digit found in string"`，这个信息就非常模糊并且没有提及无效的字符串是什么或者它正在尝试解析到哪一类整数类型。如果我们正在调试这个问题，这个错误信息几乎没什么用。尽管如此，我们还可以自己动手提供所有的上下文信息来改善这个问题：

```

sum += line.parse::<i32>()
    .map_err(|_| format!("failed to parse {} into i32", line))?;

```

第二种方式则是充分利用标准库中的 generic blanket impl：

```
impl<E: error::Error> From<E> for Box<dyn error::Error>;
```

这意味着，任意的 `Error` 类型都可以通过 `?` 被隐式地转换为 `Box<dyn error::Error>`，因此我们可以把任何可能产生错误的函数返回的 `Result` 中的错误类型设置为 `Box<dyn error::Error>`，这样 `?` 操作符就可以帮我们完成剩下的工作：

```

use std::fs::File;
use std::io::Read;
use std::path::Path;
use std::error;

fn sum_file(path: &Path) -> Result<i32, Box<dyn error::Error>> {
    letmut file = File::open(path)?; // ↑ io::Error -> Box<dyn error::Error>
    letmut contents = String::new();
    file.read_to_string(&mut contents)?; // ↑ io::Error -> Box<dyn error::Error>
    letmut sum = 0;
    for line in contents.lines() {
        sum += line.parse::<i32>()?; // ↑ ParseIntError -> Box<dyn error::Error>
    }
    Ok(sum)
}

```

虽然更为简洁，但是它似乎也存在着前面一种方式的缺点，即丢掉了类型信息。大多数情况下的确如此，但是如果调用者知道函数的实现细节，它们仍然可以通过使用 `error::Error` 上的 `downcast_ref()` 方法来处理不同的错误类型，这与它在 `dyn Any` 类型上的作用相同。

```

fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("the sum is {}", sum),
        Err(err) => {
            ifletSome(e) = err.downcast_ref::<io::Error>() {
                // 处理 io::Error
            } elseifletSome(e) = err.downcast_ref::<ParseIntError>() {
                // 处理 ParseIntError
            } else {
                // 我们知道 sum_file 只会返回上面错误中的其中一个
                // 所以不会到达这个分支
                unreachable!();
            }
        }
    }
}

```

第三种方法是最稳健和类型安全的方法，它可以汇总这些不同的错误，使用一个枚举类型构建我们自己的自定义错误类型：

```

use std::num::ParseIntError;
use std::fs::File;
use std::io;
use std::io::Read;
use std::path::Path;
use std::error;
use std::fmt;

#[derive(Debug)]
enum SumFileError {
    Io(io::Error),
    Parse(ParseIntError),
}

implFrom<io::Error> for SumFileError {
    fn from(err: io::Error) -> Self {
        SumFileError::Io(err)
    }
}

implFrom<ParseIntError> for SumFileError {
    fn from(err: ParseIntError) -> Self {
        SumFileError::Parse(err)
    }
}


impl fmt::Display for SumFileError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        matchself {
            SumFileError::Io(err) => write!(f, "sum file error: {}", err),
            SumFileError::Parse(err) => write!(f, "sum file error: {}", err),
        }
    }
}

impl error::Error for SumFileError {
    // 这个方法的默认实现总是返回 None
    //但是我们现在重写它，让它更有用
    fn source(&self) -> Option<&(dyn error::Error + 'static)> {
        Some(matchself {
            SumFileError::Io(err) => err,
            SumFileError::Parse(err) => err,
        })
    }
}

fn sum_file(path: &Path) -> Result<i32, SumFileError> {
    letmut file = File::open(path)?; // 📖 io::Error -> SumFileError
    letmut contents = String::new();
    file.read_to_string(&mut contents)?; // 📖 io::Error -> SumFileError
    letmut sum = 0;

```

```

for line in contents.lines() {
    sum += line.parse::<i32>()?; //  ParseIntError -> SumFileError
}
Ok(sum)
}

fn handle_sum_file_errors(path: &Path) {
    match sum_file(path) {
        Ok(sum) => println!("the sum is {}", sum),
        Err(SumFileError::Io(err)) => {
            // 处理 io::Error
        },
        Err(SumFileError::Parse(err)) => {
            // 处理 ParseIntError
        },
    }
}

```

继续转换类型（Conversion Traits Continued）

TryFrom & TryInto

TryFrom 和 **TryInto** 是 **From** 和 **Into** 的可能会失败的版本。

```

trait TryFrom<T> {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

trait TryInto<T> {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}

```

类似于 **Into**，我们无法实现 **TryInto**，因为它的实现是由 generic blanket impl 提供：

```
impl<T, U> TryInto<U> for T
where
    U: TryFrom<T>,
{
    type Error = U::Error;

    fn try_into(self) -> Result<U, U::Error> {
        U::try_from(self)
    }
}
```

假定在我们的程序上下文环境中，`Point` 中的 `x` 和 `y` 如果值小于 `-1000` 或者大于 `1000` 没有意义。下面是我们使用 `TryFrom` 重写之前的 `From` 实现来告诉用户，现在这种转换可以失败。

```

use std::convert::TryFrom;
use std::error;
use std::fmt;

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

#[derive(Debug)]
struct OutOfBounds;

impl fmt::Display for OutOfBounds {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "out of bounds")
    }
}

impl error::Error for OutOfBounds {}

// 现在是可以出错的
impl TryFrom<(i32, i32)> for Point {
    type Error = OutOfBounds;
    fn try_from((x, y): (i32, i32)) -> Result<Point, OutOfBounds> {
        if x.abs() > 1000 || y.abs() > 1000 {
            return Err(OutOfBounds);
        }
        Ok(Point { x, y })
    }
}

// 仍然是不会出错的
impl From<Point> for (i32, i32) {
    fn from(Point { x, y }: Point) -> Self {
        (x, y)
    }
}

```

下面是对 `Triangle` 的 `TryFrom<[TryInto<Point>; 3]>` 实现：


```

use std::convert::{TryFrom, TryInto};
use std::error;
use std::fmt;

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

#[derive(Debug)]
struct OutOfBounds;

impl fmt::Display for OutOfBounds {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "out of bounds")
    }
}

impl error::Error for OutOfBounds {}

impl TryFrom<(i32, i32)> for Point {
    type Error = OutOfBounds;
    fn try_from((x, y): (i32, i32)) -> Result<Self, Self::Error> {
        if x.abs() > 1000 || y.abs() > 1000 {
            return Err(OutOfBounds);
        }
        Ok(Point { x, y })
    }
}

struct Triangle {
    p1: Point,
    p2: Point,
    p3: Point,
}

impl<P> TryFrom<[P; 3]> for Triangle
where
    P: TryInto<Point>,
{
    type Error = P::Error;
    fn try_from([p1, p2, p3]: [P; 3]) -> Result<Self, Self::Error> {
        Ok(Triangle {
            p1: p1.try_into()?,
            p2: p2.try_into()?,
            p3: p3.try_into()?,
        })
    }
}

fn example() -> Result<Triangle, OutOfBounds> {
    let t: Triangle = [(0, 0), (1, 1), (2, 2)].try_into()?;
    Ok(t)
}

```

```
}
```

FromStr

```
trait FromStr {  
    type Err;  
    fn from_str(s: &str) -> Result<Self, Self::Err>;  
}
```

FromStr 类型允许执行一个从 **&str** 到 **Self** 的可失败的转换。最常见的使用是在 **&str** 上调用 **.parse()** 方法：

```
use std::str::FromStr;  
  
fn example<T: FromStr>(s: &'static str) {  
    // 这些都是相等的  
    let t: Result<T, _> = FromStr::from_str(s);  
    let t = T::from_str(s);  
    let t: Result<T, _> = s.parse();  
    let t = s.parse::<T>(); // 最常见的  
}
```

例如，在 **Point** 上的实现：

```

use std::error;
use std::fmt;
use std::iter::Enumerate;
use std::num::ParseIntError;
use std::str::{Chars, FromStr};

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

impl Point {
    fn new(x: i32, y: i32) -> Self {
        Point { x, y }
    }
}

#[derive(Debug, PartialEq)]
struct ParsePointError;

impl fmt::Display for ParsePointError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "failed to parse point")
    }
}

impl From<ParseIntError> for ParsePointError {
    fn from(_e: ParseIntError) -> Self {
        ParsePointError
    }
}

impl error::Error for ParsePointError {}

impl FromStr for Point {
    type Err = ParsePointError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let is_num = |(c, _): &(usize, char)| matches!(c, '0'..'9' | '-');
        let isnt_num = |t: &(_, _)| !is_num(t);

        let get_num =
            |char_idxs: &mut Enumerate<Chars<'_>>| -
            > Result<(usize, usize), ParsePointError> {
            let (start, _) = char_idxs
                .skip_while(isnt_num)
                .next()
                .ok_or(ParsePointError)?;
            let (end, _) = char_idxs

```


```


                .skip_while(is_num)
                .next()
                .ok_or(ParsePointError)?;
Ok((start, end))
    };


letmut char_idx = s.chars().enumerate();
let (x_start, x_end) = get_num(&mut char_idx)?;
let (y_start, y_end) = get_num(&mut char_idx)?;

let x = s[x_start..x_end].parse::<i32>()?;
let y = s[y_start..y_end].parse::<i32>()?;

Ok(Point { x, y })
}

#[test]// 
fn pos_x_y() {
    let p = "(4, 5)".parse::<Point>();
    assert_eq!(p, Ok(Point::new(4, 5)));
}

#[test]// 
fn neg_x_y() {
    let p = "(-6, -2)".parse::<Point>();
    assert_eq!(p, Ok(Point::new(-6, -2)));
}

#[test] // 
fn not_a_point() {
    let p = "not a point".parse::<Point>();
    assert_eq!(p, Err(ParsePointError));
}

```

FromStr 和 **TryFrom<&str>** 有着相同的签名。只要我们通过其中一个实现另一个，先实现哪个并不重要。下面是对 **Point** 实现 **TryFrom<&str>**，假定它已经实现了 **FromStr**：

```

impl TryFrom<&str> for Point {
    type Error = <Point as FromStr>::Error;
    fn try_from(s: &str) -> Result<Point, Self::Error> {
        <Point as FromStr>::from_str(s)
    }
}

```

AsRef & AsMut

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) -> &T;
}
```

```
trait AsMut<T: ?Sized> {
    fn as_mut(&mutself) -> &mut T;
}
```

AsRef 被用于轻量级的引用到引用之间的转换。然而，它最常见的一个用途是使函数在是否获取所有权上具有通用性：

```
// 接受：
// - &str
// - &String
fn takes_str(s: &str) {
    // use &str
}
```

```
// 接受：
// - &str
// - &String
// - String
fn takes_asref_str<S: AsRef<str>>(s: S) {
    let s: &str = s.as_ref();
    // 使用 &str
}
```

```
fn example(slice: &str, borrow: &String, owned: String) {
    takes_str(slice);
    takes_str(borrow);
    takes_str(owned); // ✗
    takes_asref_str(slice);
    takes_asref_str(borrow);
    takes_asref_str(owned); // ✓
}
```

另一个常见用途是返回一个内部私有数据的引用，该数据由一个保护不变性的类型所包裹。标准库中一个比较好的示例是 **String**，它包裹了 **Vec<u8>**：

```
struct String {
    vec: Vec<u8>,
}
```

内部的 **Vec<u8>** 不能被公开，因为如果这样的话，人们就会修改里面的字节并破坏 **String** 中有效的 UTF-8 编码。但是，暴露内部字节数组的一个不可变的只读引用是安全的，即下面的实现：

```
impl AsRef<[u8]> for String;
```

一般而言，只有当一个类型包裹了其他类型用来为该内部类型提供了额外功能或者保护内部类型的不变性时，为这样的类型实现 **AsRef** 才有意义。让我们来看一个 **AsRef** 的不合适使用：

```
struct User {
    name: String,
    age: u32,
}

impl AsRef<String> for User {
    fn as_ref(&self) -> &String {
        &self.name
    }
}

impl AsRef<u32> for User {
    fn as_ref(&self) -> &u32 {
        &self.age
    }
}
```

一开始是可行的，而且看上去还有点道理，但是当我们为 **User** 添加更多成员时，问题就出现了：

```
struct User {
    name: String,
    email: String,
    age: u32,
    height: u32,
}

impl AsRef<String> for User {
    fn as_ref(&self) -> &String {
        //我们返回 name 还是 email?
    }
}

impl AsRef<u32> for User {
    fn as_ref(&self) -> &u32 {
        //我们返回 age 还是 height?
    }
}
```

User 是由 **String** 和 **u32** 组成，但是它并不等同于一个 **String** 和一个 **u32**，甚至我们还会有更多的类型：

```
struct User {  
    name: Name,  
    email: Email,  
    age: Age,  
    height: Height,  
}
```

对于这样的类型实现 `AsRef` 没有什么意义，因为 `AsRef` 用于语义相等的事物之间引用到引用的转换，而且 `Name` 、 `Email` 、 `Age` 以及 `Height` 并不等同于一个 `User` 。

下面是一个好的示例，其中，我们会引入一个新类型 `Moderator` ，它只包裹了一个 `User` 并添加了特定的审核权限：

```

struct User {
    name: String,
    age: u32,
}

//不幸地是，标准库并没有提供一个generic blanket impl来避免这种重复的实现
impl AsRef<User> for User {
    fn as_ref(&self) -> &User {
        self
    }
}

enum Privilege {
    BanUsers,
    EditPosts,
    DeletePosts,
}

//尽管 Moderators 有一些特殊权限，它们仍然是普通的 User
//并且应该做相同的事情
struct Moderator {
    user: User,
    privileges: Vec<Privilege>
}

impl AsRef<Moderator> for Moderator {
    fn as_ref(&self) -> &Moderator {
        self
    }
}

impl AsRef<User> for Moderator {
    fn as_ref(&self) -> &User {
        &self.user
    }
}

//使用 User 和 Moderators （也是一种User）应该都是可以调用的
fn create_post<U: AsRef<User>>(u: U) {
    let user = u.as_ref();
    // etc
}

fn example(user: User, moderator: Moderator) {
    create_post(&user);
    create_post(&moderator); // ✅
}

```


这是有效的，因为 `Moderator` 就是 `User`。下面是 `Deref` 章节中的例子，我们用了 `AsRef` 来实现：

```
use std::convert::AsRef;

struct Human {
    health_points: u32,
}

implAsRef<Human> for Human {
    fn as_ref(&self) -> &Human {
        self
    }
}

enum Weapon {
    Spear,
    Axe,
    Sword,
}

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

implAsRef<Soldier> for Soldier {
    fn as_ref(&self) -> &Soldier {
        self
    }
}

implAsRef<Human> for Soldier {
    fn as_ref(&self) -> &Human {
        &self.human
    }
}

enum Mount {
    Horse,
    Donkey,
    Cow,
}

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

implAsRef<Knight> for Knight {
    fn as_ref(&self) -> &Knight {
        self
    }
}
```

```
implAsRef<Soldier> for Knight {
fn as_ref(&self) -> &Soldier {
    &self.soldier
}
}

implAsRef<Human> for Knight {
fn as_ref(&self) -> &Human {
    &self.soldier.human
}
}

enum Spell {
    MagicMissile,
    FireBolt,
    ThornWhip,
}

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

implAsRef<Mage> for Mage {
fn as_ref(&self) -> &Mage {
    self
}
}

implAsRef<Human> for Mage {
fn as_ref(&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 AsRef<Wizard> for Wizard {
    fn as_ref(&self) -> &Wizard {
        self
    }
}
```

```
impl AsRef<Mage> for Wizard {
    fn as_ref(&self) -> &Mage {
        &self.mage
    }
}
```

```
impl AsRef<Human> for Wizard {
    fn as_ref(&self) -> &Human {
        &self.mage.human
    }
}
```

```
fn borrows_human<H: AsRef<Human>>(human: H) {}
fn borrows_soldier<S: AsRef<Soldier>>(soldier: S) {}
fn borrows_knight<K: AsRef<Knight>>(knight: K) {}
fn borrows_mage<M: AsRef<Mage>>(mage: M) {}
fn borrows_wizard<W: AsRef<Wizard>>(wizard: W) {}
```

```
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);
}
```

Deref 在之前的例子中没有起作用，是因为解引用强制转换是类型间的隐式转换，这就为人们制定错误的想法并对其行为方式的期望留下了空间。**AsRef** 能够工作是因为它让类型之间的转换变为显式的，并且没有给开发者错误的想法和期望留有余地。

Borrow & BorrowMut

```
trait Borrow<Borrowed>
where
    Borrowed: ?Sized,
{
    fn borrow(&self) -> &Borrowed;
}

trait BorrowMut<Borrowed>: Borrow<Borrowed>
where
    Borrowed: ?Sized,
{
    fn borrow_mut(&mutself) -> &mut Borrowed;
}
```

这些 `trait` 被发明用于解决非常具体的问题，即使用 `&str` 类型的值在 `HashSet`、`HashMap`、`BTreeSet` 和 `BTreeMap` 中查找 `String` 类型的 `key`。

我们可以把 `Borrow<T>` 和 `BorrowMut<T>` 看作更严格的 `AsRef<T>` 和 `AsMut<T>`，它们返回的引用 `&T` 与 `Self` 有等价性的 `Eq`、`Hash` 和 `Ord` 实现。通过下面的例子会更易于理解：

```
use std::borrow::Borrow;
use std::hash::Hasher;
use std::collections::hash_map::DefaultHasher;
use std::hash::Hash;
```

```
fn get_hash<T: Hash>(t: T) -> u64 {
    let mut hasher = DefaultHasher::new();
    t.hash(&mut hasher);
    hasher.finish()
}
```

```
fn asref_example<Owned, Ref>(owned1: Owned, owned2: Owned)
where
    Owned: Eq + Ord + Hash + AsRef<Ref>,
    Ref: Eq + Ord + Hash
{
    let ref1: &Ref = owned1.as_ref();
    let ref2: &Ref = owned2.as_ref();
```

```
// refs aren't required to be equal if owned types are equal
assert_eq!(owned1 == owned2, ref1 == ref2); // ❌
```

```
let owned1_hash = get_hash(&owned1);
let owned2_hash = get_hash(&owned2);
let ref1_hash = get_hash(&ref1);
let ref2_hash = get_hash(&ref2);
```

```
// ref hashes aren't required to be equal if owned type hashes are equal
assert_eq!(owned1_hash == owned2_hash, ref1_hash == ref2_hash); // ❌
```

```
// ref comparisons aren't required to match owned type comparisons
assert_eq!(owned1.cmp(&owned2), ref1.cmp(&ref2)); // ❌
}
```

```
fn borrow_example<Owned, Borrowed>(owned1: Owned, owned2: Owned)
where
    Owned: Eq + Ord + Hash + Borrow<Borrowed>,
    Borrowed: Eq + Ord + Hash
{
    let borrow1: &Borrowed = owned1.borrow();
    let borrow2: &Borrowed = owned2.borrow();
```

```
// borrows are required to be equal if owned types are equal
assert_eq!(owned1 == owned2, borrow1 == borrow2); // ✅
```

```

let owned1_hash = get_hash(&owned1);
let owned2_hash = get_hash(&owned2);
let borrow1_hash = get_hash(&borrow1);
let borrow2_hash = get_hash(&borrow2);

// borrow hashes are required to be equal if owned type hashes are equal
assert_eq!(owned1_hash == owned2_hash, borrow1_hash == borrow2_hash); // ✓

// borrow comparisons are required to match owned type comparisons
assert_eq!(owned1.cmp(&owned2), borrow1.cmp(&borrow2)); // ✓
}

```

意识到这些 trait 以及它们为什么存在是有益的，因为它有助于搞清楚 `HashSet`、`HashMap`、`BTreeSet` 以及 `BTreeMap` 的某些方法，但是我们很少需要为我们的类型实现这些 trait，因为我们很少需要创建一对儿类型，其中一个是另一个的借用版本。如果我们有某个类型 `T`，`&T` 在 99.99% 的情况下可以完成工作，并且因为 generic blanket impl，`T: Borrow<T>` 已经为所有的类型 `T` 实现了，所以我们不需要手动地实现它并且我们不需要创建一个 `U` 以用来 `T: Borrow<U>`。

ToOwned

```

trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;

    // 提供默认实现
    fn clone_into(&self, target: &mut Self::Owned);
}

```

`ToOwned` 是 `Clone` 的一个更为通用的版本。`Clone` 允许我们获取一个 `&T` 并把它转为一个 `T`，但是 `ToOwned` 允许我们拿到一个 `&Borrowed` 并把它转为一个 `Owned`，其中 `Owned: Borrow<Borrowed>`。

换句话说，我们不能从一个 `&str` 克隆一个 `String`，或者从一个 `&Path` 克隆一个 `PathBuf`，或者从一个 `&OsStr` 克隆一个 `OsString`，因为 `clone` 方法签名不支持这种跨类型的克隆，这就是 `ToOwned` 产生的原因。

类似于 `Borrow` 和 `BorrowMut`，知道这个 trait 并理解它为什么存在同样是有益的，只是我们几乎不需要为我们的类型实现它。

