


【译】Rust标准库Trait指南（一）

 mp.weixin.qq.com/s/kY9q7_mv3sfGq8kPM0_meg

原创 Rust碎碎念 Rust碎碎念 5月1日

收录于话题

#Rust翻译文章集合 60 个内容


#Rust 14 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#intro>

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 
 - Self 
 - 函数 
 - 方法 
 - 关联类型 
 - 泛型类型 
 - 泛型类型 vs 关联类型 
 - Trait 项 
 - 作用域 
 - 派生宏 
 - 默认实现 (Default Impls) 
 - 泛型覆盖实现 
 - Subtraits & SuperTraits 
 - Trait 对象 
 - 标记 Trait 
 - 自动 Trait 
 - 不安全 Traits 
- 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 

引言

你是否曾想过下面这些 trait 有什么不同？

- `Deref<T>` , `AsRef<T>` , 以及 `Borrow<T>` ?
- `Clone` , `Copy` , 和 `ToOwned` ?
- `From<T>` 和 `Into<T>` ?
- `TryFrom<&str>` 和 `FromStr` ?
- `FnOnce` , `FnMut` , `Fn` 和 `fn` ?

或者你曾问过自己下面这些问题：

- “我在 trait 中，什么时候使用关联类型（associated type），什么时候使用泛型（generic types）？”
- “什么是泛型覆盖实现（generic blanket impls^[1]）？”
- “subtrait 和 supertrait 是如何工作的？”
- “为什么这个 trait 没有任何方法？”

那么这篇文章就是为你而写的！它回答了包括但不限于上述所有的问题。我们将一起对 Rust 标准库中所有最流行和最常用的 trait 进行快速的浏览。

你可以按章节顺序阅读本文，也可以跳到你最感兴趣的 trait，因为每个 trait 章节的开头都有一个指向前置章节的链接列表，你应该阅读这些链接，以便有足够的背景知识来理解当前章节的解释（译注：很抱歉，译文中暂时无法提供链接跳转）。

Trait 基础

我们将会覆盖足够多的基础知识，这样文章的其余部分就可以精简，而不必因为它们在不同的 trait 中反复出现而重复解释相同的概念。

Trait 项 (Item)

Trait 项是指包含于 trait 声明中的任意项。

Self

`Self` 总是指代实现类型。

```

trait Trait {
    // always returns i32
    fn returns_num() -> i32;

    // returns implementing type
    fn returns_self() -> Self;
}

struct SomeType;
struct OtherType;

impl Trait for SomeType {
    fn returns_num() -> i32 {
        5
    }

    // Self == SomeType
    fn returns_self() -> Self {
        SomeType
    }
}

impl Trait for OtherType {
    fn returns_num() -> i32 {
        6
    }

    // Self == OtherType
    fn returns_self() -> Self {
        OtherType
    }
}

```

函数 (Function)

Trait 函数是指第一个参数不是 `self` 关键字的任意函数。

```

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

```

Trait 函数可以通过 trait 或者实现类型的命名空间来调用。

```

fn main() {
    let zero: i32 = Default::default();
    let zero = i32::default();
}

```

方法 (Method)

Trait 方法是指，第一个参数使用了 `self` 关键字并且 `self` 的类型是 `Self` , `&Self` , `&mut Self` 之一。 `self` 的类型也可以被 `Box` , `Rc` , `Arc` 或 `Pin` 来包装。

```
trait Trait {  
    // methods  
    fn takes_self(self);  
    fn takes_immut_self(&self);  
    fn takes_mut_self(&mutself);  
  
    // above methods desugared  
    fn takes_self(self: Self);  
    fn takes_immut_self(self: &Self);  
    fn takes_mut_self(self: &mutSelf);  
}  
  
    // example from standard library  
trait ToString {  
    fn to_string(&self) -> String;  
}
```

Trait 方法可以通过在实现类型上使用点 (.) 操作符来调用。

```
fn main() {  
    let five = 5.to_string();  
}
```

此外，trait 方法还可以像函数那样由 trait 或者实现类型通过命名空间来调用。

```
fn main() {  
    let five = ToString::to_string(&5);  
    let five = i32::to_string(&5);  
}
```

关联类型 (Associated Types)

Trait 可以有关联类型。当我们需要在函数签名中使用 `Self` 以外的某个类型，但是希望这个类型可以由实现者来选择而不是硬编码到 trait 声明中，这时关联类型就可以发挥作用了。

```
trait Trait {
    type AssociatedType;
    fn func(arg: Self::AssociatedType);
}

struct SomeType;
struct OtherType;

// any type implementing Trait can
// choose the type of AssociatedType

impl Trait for SomeType {
    type AssociatedType = i8; // chooses i8
    fn func(arg: Self::AssociatedType) {}
}

impl Trait for OtherType {
    type AssociatedType = u8; // chooses u8
    fn func(arg: Self::AssociatedType) {}
}

fn main() {
    SomeType::func(-1_i8); // can only call func with i8 on SomeType
    OtherType::func(1_u8); // can only call func with u8 on OtherType
}
```

泛型参数（Generic Parameters）

“泛型参数”泛指泛型类型参数（generic type parameters）、泛型生命周期参数（generic lifetime parameters）、以及泛型常量参数（generic const parameters）。因为这些说起来比较拗口，所以人们通常把它们简称为“泛型类型（generic type）”、“生命周期（lifetime）”和“泛型常量（generic const）”。由于我们将要讨论的所有标准库 trait 中都没有使用泛型常量，所以它们不在本文的讨论范围之内。

我们可以使用参数来对一个 trait 声明进行泛化（generalize）。

```
// trait declaration generalized with lifetime & type parameters
trait Trait<'a, T> {
// signature uses generic type
fn func1(arg: T);

// signature uses lifetime
fn func2(arg: &'ai32);

// signature uses generic type & lifetime
fn func3(arg: &'a T);
}

struct SomeType;

impl<'a> Trait<'a, i8> for SomeType {
fn func1(arg: i8) {}
fn func2(arg: &'ai32) {}
fn func3(arg: &'ai8) {}
}

impl<'b> Trait<'b, u8> for SomeType {
fn func1(arg: u8) {}
fn func2(arg: &'b i32) {}
fn func3(arg: &'b u8) {}
}
```

泛型可以具有默认值，最常用的默认值是 `Self`，但是任何类型都可以作为默认值。

```
// make T = Self by default
trait Trait<T = Self> {
    fn func(t: T) {}
}

// any type can be used as the default
trait Trait2<T = i32> {
    fn func2(t: T) {}
}

struct SomeType;

// omitting the generic type will
// cause the impl to use the default
// value, which is Self here
impl Trait for SomeType {
    fn func(t: SomeType) {}
}

// default value here is i32
impl Trait2 for SomeType {
    fn func2(t: i32) {}
}

// the default is overridable as we'd expect
impl Trait<String> for SomeType {
    fn func(t: String) {}
}

// overridable here too
impl Trait2<String> for SomeType {
    fn func2(t: String) {}
}
```

除了可以对 trait 进行参数化之外，我们还可以对单个函数和方法进行参数化。

```
trait Trait {
    fn func<'a, T>(t: &'a T);
}
```

泛型类型 vs 关联类型

泛型类型和关联类型都把在 trait 的函数和方法中使用哪种具体类型的决定权交给了实现者，因此这部分内容要去解释什么时候使用泛型类型，什么时候使用关联类型。

通常的经验法则是：

- 当每个类型只应该有 trait 的一个实现时，使用关联类型。
- 当每个类型可能会有 trait 的多个实现时，使用泛型类型。

比如说我们想要定义一个名为 **Add** 的 trait，该 trait 允许我们对值进行相加。下面是一个最初的设计和实现，里面只使用了关联类型。

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

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

```
impl Add for Point {  
    type Rhs = 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: 1 };  
    let p2 = Point { x: 2, y: 2 };  
    let p3 = p1.add(p2);  
    assert_eq!(p3.x, 3);  
    assert_eq!(p3.y, 3);  
}
```

假设现在我们要添加这样一种功能：把 **i32** 加到 **Point** 上，其中 **Point** 里面的成员 **x** 和 **y** 都会加上 **i32**。


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

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

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

```
impl Add for Point { // ❌
    type Rhs = i32;
    type Output = Point;
    fn add(self, rhs: i32) -> Point {
        Point {
            x: self.x + rhs,
            y: self.y + rhs,
        }
    }
}
```

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

    let p1 = Point { x: 1, y: 1 };
    let int2 = 2;
    let p3 = p1.add(int2); // ❌
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);
}
```

上面的代码会抛出错误：

```

error[E0119]: conflicting implementations of trait `Add` for type `Point`:
  --> src/main.rs:23:1
   |
12 | impl Add for Point {
   | ----- first implementation here
...
23 | impl Add for Point {
   | ~~~~~~ conflicting implementation for `Point`

```

因为 **Add** trait 没有被任何的泛型类型参数化，我们只能在每个类型上实现这个 trait 一次，这意味着，我们只能一次把 **Rhs** 和 **Output** 类型都选取好！为了能够使 **Point** 和 **i32** 类型都能和 **Point** 相加，我们必须把 **Rhs** 从一个关联类型重构为泛型类型，这样就能够让我们根据 **Rhs** 不同的类型参数来为 **Point** 实现 trait 多次。

```

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

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

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

impl Add<i32> for Point { // ✓
    type Output = Self;
    fn add(self, rhs: i32) -> Self::Output {
        Point {
            x: self.x + rhs,
            y: self.y + rhs,
        }
    }
}

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

    let p1 = Point { x: 1, y: 1 };
    let int2 = 2;
    let p3 = p1.add(int2); // ✓
    assert_eq!(p3.x, 3);
    assert_eq!(p3.y, 3);
}

```

假如说我们增加了一个名为 **Line** 的新类型，它包含两个 **Point**，现在，在我们的程序中存在这样一种上下文环境，即将两个 **Point** 相加之后应该产生一个 **Line** 而不是另一个 **Point**。这在当我们当前的 **Add** trait 设计中是不可行的，因为 **Output** 是一个关联类型，但是我们通过把 **Output** 从关联类型重构为泛型类型来实现这个新需求。


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

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

```
impl Add<Point, Point> for Point {  
    fn add(self, rhs: Point) -> Point {  
        Point {  
            x: self.x + rhs.x,  
            y: self.y + rhs.y,  
        }  
    }  
}
```

```
impl Add<i32, Point> for Point {  
    fn add(self, rhs: i32) -> Point {  
        Point {  
            x: self.x + rhs,  
            y: self.y + rhs,  
        }  
    }  
}
```

```
struct Line {  
    start: Point,  
    end: Point,  
}
```

```
impl Add<Point, Line> for Point { //   
    fn add(self, rhs: Point) -> Line {  
        Line {  
            start: self,  
            end: rhs,  
        }  
    }  
}
```

```
fn main() {  
    let p1 = Point { x: 1, y: 1 };  
    let p2 = Point { x: 2, y: 2 };  
    let p3: Point = p1.add(p2);  
    assert!(p3.x == 3 && p3.y == 3);  
}
```

```

let p1 = Point { x: 1, y: 1 };
let int2 = 2;
let p3 = p1.add(int2);
assert!(p3.x == 3 && p3.y == 3);

        let p1 = Point { x: 1, y: 1 };
        let p2 = Point { x: 2, y: 2 };
        let l: Line = p1.add(p2); // ✓
        assert!(l.start.x == 1 && l.start.y == 1 && l.end.x == 2 && l.end.y == 2)
    }

```

所以，哪个 **Add** trait 是最好的呢？这取决于你程序中的需求！放在合适的场景中，它们都很好。

作用域 (Scope)

只有当 trait 在作用域之中时，trait 项才能被使用。大多数 Rustaceans 在第一次尝试写一个 I/O 相关的程序时，都会在吃过一番苦头之后了解到这一点，因为 **Read** 和 **Write** 的 trait 并不在标准库的预置 (prelude) 中。

```

use std::fs::File;
use std::io;

fn main() -> Result<(), io::Error> {
    let mut file = File::open("Cargo.toml")?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?; // ✗ read_to_string not found in File
    Ok(())
}

```

read_to_string(buf: &mut String) 声明于 **std::io::Read** 中并且被 **std::fs::File** 结构体实现，但是要想调用它，**std::io::Read** 必须在当前作用域中。

```

use std::fs::File;
use std::io;
use std::io::Read; // ✓

fn main() -> Result<(), io::Error> {
    let mut file = File::open("Cargo.toml")?;
    let mut buffer = String::new();
    file.read_to_string(&mut buffer)?; // ✓
    Ok(())
}

```

标准库预置 (The standard library prelude) 是标准库中的一个模块，也就是说，**std::prelude::v1**，它在每个其他模块的顶部被自动导入，即 **use std::prelude::v1::***。这样的话，下面这些 trait 就总会在作用域中，我们不需要自己显式地导入它们，因为它们是预置的一部分。

- AsMut
- AsRef

- Clone
- Copy
- Default
- Drop
- Eq
- Fn
- FnMut
- FnOnce
- From
- Into
- ToOwned
- IntoIterator
- Iterator
- PartialEq
- PartialOrd
- Send
- Sized
- Sync
- ToString
- Ord

派生宏（Derive Macros）

标准库导出了一小部分派生宏，这么派生宏可以让我们可以便捷地在一个类型上实现 trait，前提是该类型的所有成员都实现了这个 trait。派生宏以它们所实现的 trait 来命名。

- Clone
- Copy
- Debug
- Default
- Eq
- Hash
- Ord
- PartialEq
- PartialOrd

使用示例：

```
// macro derives Copy & Clone impl for SomeType
#[derive(Copy, Clone)]
struct SomeType;
```

注意：派生宏也是过程宏（procedural macros），它们可以被用来做任何事情，没有强制规定它们必须要实现一个 trait，或者它们只能在所有成员都实现 trait 的情况下才能工作，这些只是标准库中派生宏所遵循的惯例。

默认实现（Default Impls）

Trait 可以为它们的函数和方法提供默认实现。

```
trait Trait {
    fn method(&self) {
        println!("default impl");
    }
}

struct SomeType;
struct OtherType;

// use default impl for Trait::method
impl Trait for SomeType {}

impl Trait for OtherType {
    // use our own impl for Trait::method
    fn method(&self) {
        println!("OtherType impl");
    }
}

fn main() {
    SomeType.method(); // prints "default impl"
    OtherType.method(); // prints "OtherType impl"
}
```

如果 trait 中的某些方法是完全通过 trait 的另一些方法来实现的，这就非常方便了。

```

trait Greet {
    fn greet(&self, name: &str) -> String;
    fn greet_loudly(&self, name: &str) -> String {
        self.greet(name) + "!"
    }
}

struct Hello;
struct Hola;

impl Greet for Hello {
    fn greet(&self, name: &str) -> String {
        format!("Hello {}", name)
    }
    // use default impl for greet_loudly
}

impl Greet for Hola {
    fn greet(&self, name: &str) -> String {
        format!("Hola {}", name)
    }
    // override default impl
    fn greet_loudly(&self, name: &str) -> String {
        letmut greeting = self.greet(name);
        greeting.insert_str(0, "i");
        greeting + "!"
    }
}

fn main() {
    println!("{}", Hello.greet("John")); // prints "Hello John"
    println!("{}", Hello.greet_loudly("John")); // prints "Hello John!"
    println!("{}", Hola.greet("John")); // prints "Hola John"
    println!("{}", Hola.greet_loudly("John")); // prints "iHola John!"
}

```

标准库中的很多 trait 为很多它们的方法提供了默认实现。

泛型覆盖实现（Generic Blanket Impls）

泛型覆盖实现是一种在泛型类型而不是具体类型上的实现，为了解释为什么以及如何使用它，让我们从为整数类型实现一个 `is_even` 方法开始。



```
trait Even {
    fn is_even(self) -> bool;
}

impl Even for i8 {
    fn is_even(self) -> bool {
        self % 2_i8 == 0_i8
    }
}

impl Even for u8 {
    fn is_even(self) -> bool {
        self % 2_u8 == 0_u8
    }
}

impl Even for i16 {
    fn is_even(self) -> bool {
        self % 2_i16 == 0_i16
    }
}

// etc

#[test] // 
fn test_is_even() {
    assert!(2_i8.is_even());
    assert!(4_u8.is_even());
    assert!(6_i16.is_even());
    // etc
}
```


很明显，上面的实现十分啰嗦。而且，所有我们的实现几乎都是一样的。此外，如果 Rust 决定在未来增加更多的整数类型，我们必须回到这段代码中，用新的整数类型来更新它。我们可以通过使用泛型覆盖实现来解决所有的问题。

```

use std::fmt::Debug;
use std::convert::TryInto;
use std::ops::Rem;

trait Even {
    fn is_even(self) -> bool;
}

// generic blanket impl
impl<T> Even for T
where
    T: Rem<Output = T> + PartialEq<T> + Sized,
    u8: TryInto<T>,
    <u8 as TryInto<T>>::Error: Debug,
{
    fn is_even(self) -> bool {
        // these unwraps will never panic
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
    }
}

#[test] // 
fn test_is_even() {
    assert!(2_i8.is_even());
    assert!(4_u8.is_even());
    assert!(6_i16.is_even());
    // etc
}

```

不同于默认实现，泛型覆盖实现提供了方法的实现，所以它们不能被重写。

```

use std::fmt::Debug;
use std::convert::TryInto;
use std::ops::Rem;

trait Even {
    fn is_even(self) -> bool;
}

impl<T> Even for T
where
    T: Rem<Output = T> + PartialEq<T> + Sized,
    u8: TryInto<T>,
    <u8 as TryInto<T>>::Error: Debug,
{
    fn is_even(self) -> bool {
        self % 2.try_into().unwrap() == 0.try_into().unwrap()
    }
}

impl Even for u8 { // ❌
    fn is_even(self) -> bool {
        self % 2_u8 == 0_u8
    }
}

```

上面的代码会抛出下面的错误：

```

error[E0119]: conflicting implementations of trait `Even` for type `u8`:
  --> src/lib.rs:22:1
   |
10 | / impl<T> Even for T
11 | | where
12 | |     T: Rem<Output = T> + PartialEq<T> + Sized,
13 | |     u8: TryInto<T>,
... |
19 | |     }
20 | | }
   | |_- first implementation here
21 |
22 |     impl Even for u8 {
   |     ~~~~~~ conflicting implementation for `u8`

```

这些实现有重叠，因此它们是冲突的，所以 Rust 拒绝编译这段代码以确保 trait 的一致性。trait 一致性是指，对于任意给定的类型，最多存在某一 trait 的一个实现。Rust 用来强制执行特质一致性的规则，这些规则的含义，以及针对这些含义的变通方案都不在本文的讨论范围之内。

Subtraits & Supertraits

subtrait 中的 **sub** 指的是子集 (subset)，**supertrait** 中的 **super** 指的是超集 (superset)。如果我们有下面这个 trait 声明：

```
trait Subtrait: Supertrait {}
```

所有实现了 `Subtrait` 的类型是所有实现了 `Supertrait` 的类型的子集，或者反过来讲：所有实现了 `Supertrait` 的类型是所有实现了 `Subtrait` 类型的超集。而且，上面的代码是一种语法糖，展开来应该是：

```
trait Subtrait where Self: Supertrait {}
```

这是一个微妙而重要的区别，要明白约束在 `Self` 上，也就是实现 `Subtrait` 的类型而非 `Subtrait` 自身。后者也没有意义，因为 `trait` 约束只能作用于能够实现 `trait` 的具体类型，`trait` 本身不能实现其他的 `trait`：

```
trait Supertrait {
    fn method(&self) {
        println!("in supertrait");
    }
}
```

```
trait Subtrait: Supertrait {
    // this looks like it might impl or
    // override Supertrait::method but it
    // does not
    fn method(&self) {
        println!("in subtrait")
    }
}
```

```
struct SomeType;
```

```
// adds Supertrait::method to SomeType
impl Supertrait for SomeType {}
```

```
// adds Subtrait::method to SomeType
impl Subtrait for SomeType {}
```

```
// both methods exist on SomeType simultaneously
// neither overriding or shadowing the other
```

```
fn main() {
    SomeType.method(); // ❌ ambiguous method call
    // must disambiguate using fully-qualified syntax
    <SomeType as Supertrait>::method(&st); // ✅ prints "in supertrait"
    <SomeType as Subtrait>::method(&st); // ✅ prints "in subtrait"
}
```

此外，对于一个类型如何同时实现一个 `subtrait` 和一个 `supertrait`，也没有明确的规则。它可以在另一个类型的实现中实现其他的方法。

```
trait Supertrait {
    fn super_method(&mutself);
}

trait Subtrait: Supertrait {
    fn sub_method(&mutself);
}

struct CallSuperFromSub;

impl Supertrait for CallSuperFromSub {
    fn super_method(&mutself) {
        println!("in super");
    }
}

impl Subtrait for CallSuperFromSub {
    fn sub_method(&mutself) {
        println!("in sub");
        self.super_method();
    }
}

struct CallSubFromSuper;

impl Supertrait for CallSubFromSuper {
    fn super_method(&mutself) {
        println!("in super");
        self.sub_method();
    }
}

impl Subtrait for CallSubFromSuper {
    fn sub_method(&mutself) {
        println!("in sub");
    }
}

struct CallEachOther(bool);

impl Supertrait for CallEachOther {
    fn super_method(&mutself) {
        println!("in super");
        if self.0 {
            self.0 = false;
            self.sub_method();
        }
    }
}
```

```

    }
}

impl Subtrait for CallEachOther {
fn sub_method(&mutself) {
println!("in sub");
ifself.0 {
self.0 = false;
self.super_method();
}
}
}

fn main() {
    CallSuperFromSub.super_method(); // prints "in super"
    CallSuperFromSub.sub_method(); // prints "in sub", "in super"

    CallSubFromSuper.super_method(); // prints "in super", "in sub"
    CallSubFromSuper.sub_method(); // prints "in sub"

    CallEachOther(true).super_method(); // prints "in super", "in sub"
    CallEachOther(true).sub_method(); // prints "in sub", "in super"
}

```

希望上面的例子能够表达出，subtrait 和 supertrait 之间可以是很复杂的关系。在介绍能够将这些复杂性进行整洁封装的心智模型之前，让我们快速回顾并建立我们用来理解泛型类型上的 trait 约束的心智模型。

```

fn function<T: Clone>(t: T) {
    // impl
}

```

在不知道这个函数的实现的情况下，我们可以合理地猜测，`t.clone()` 会在某个时候被调用，因为当一个泛型类型被一个 trait 所约束时，意味着它对 trait 有依赖性。泛型与 trait 约束之间关系的心智模型是一个简单而直观模型：泛型依赖于 trait 约束。

现在让我们看看 `Copy` 的 trait 声明：

```

trait Copy: Clone {}

```

上面的语法看起来与在一个泛型类型上应用 trait 约束很相似，但是 `Copy` 完全不依赖于 `Clone`。之前的模型在这里没有帮助。个人认为，理解 subtrait 和 supertrait 最为简洁优雅的心智模型是：subtrait 细化（refine）了它们的 supertrait。

“细化（Refinement）”刻意保持一定的模糊性，因为它们在不同的上下文环境中会有不同的含义：

- subtrait 可能会使得 supertrait 的方法实现更为具体，快速，占用更少的内存，例如，`Copy:Clone`；

- `subtrait` 可能会对 `supertrait` 的方法实现增加额外的保证，例如：`Eq`：
`PartialEq` , `Ord`: `PartialOrd` , `ExactSizeIterator`: `Iterator` ;
- `subtrait` 可能会使得 `supertrait` 的方法更为灵活和易于调用，例如：`FnMut`：
`FnOnce` , `Fn`: `FnMut` ;
- `subtrait` 可能会扩展 `supertrait` 并添加新的方法，例如：`DoubleEndedIterator`:
`Iterator` , `ExactSizeIterator`: `Iterator` 。

Trait 对象

泛型给我们提供了编译期多态，而 `trait` 对象给我们提供了运行时多态。我们可以使用 `trait` 对象来让函数在运行时动态地返回不同的类型。

```
fn example(condition: bool, vec: Vec<i32>) -> Box<dyn Iterator<Item = i32>> {
    let iter = vec.into_iter();
    if condition {
        // Has type:
        // Box<Map<IntoIter<i32>, Fn(i32) -> i32>>
        // But is cast to:
        // Box<dyn Iterator<Item = i32>>
        Box::new(iter.map(|n| n * 2))
    } else {
        // Has type:
        // Box<Filter<IntoIter<i32>, Fn(&i32) -> bool>>
        // But is cast to:
        // Box<dyn Iterator<Item = i32>>
        Box::new(iter.filter(|&n| n >= 2))
    }
}
```

Trait 对象还允许我们在集合中存储多种类型：

```

use std::f64::consts::PI;

struct Circle {
    radius: f64,
}

struct Square {
    side: f64
}

trait Shape {
    fn area(&self) -> f64;
}

impl Shape for Circle {
    fn area(&self) -> f64 {
        PI * self.radius * self.radius
    }
}

impl Shape for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn get_total_area(shapes: Vec<Box<dyn Shape>>) -> f64 {
    shapes.into_iter().map(|s| s.area()).sum()
}

fn example() {
    let shapes: Vec<Box<dyn Shape>> = vec![
        Box::new(Circle { radius: 1.0 }), // Box<Circle> cast to Box<dyn Shape>
        Box::new(Square { side: 1.0 }), // Box<Square> cast to Box<dyn Shape>
    ];
    assert_eq!(PI + 1.0, get_total_area(shapes)); // ✅
}

```

Trait 对象是没有大小的，所以它们必须总是在一个指针后面。我们可以根据类型中 **dyn** 关键字的存在来区分具体类型和 trait 对象在类型级别上的区别。


```
struct Struct;
trait Trait {}

// regular struct
&Struct
Box<Struct>
Rc<Struct>
Arc<Struct>

// trait objects
&dyn Trait
Box<dyn Trait>
Rc<dyn Trait>
Arc<dyn Trait>
```

不是所有的 trait 都可以被转成 trait 对象。当且仅当一个 trait 满足下面这些要求时，它才是对象安全的（object-safe）：

- trait 不要求 `Self:Sized`
- trait 的所有方法都是对象安全的

当一个 trait 方法满足下面的要求时，该方法是对象安全的：

- 方法要求 `Self:Sized` 或者
- 方法在其接收者位置仅使用一个 `Self` 类型

理解为什么要求是这样的，与本文的其余部分无关，但如果你仍然好奇，可以阅读 [Sizeness in Rust^{\[2\]}](#)（译注：Sizedness in Rust 这篇文章已翻译，见下方链接）。

【译】Rust中的Sizeness

标记 Trait（Marker Traits）

标记 trait 是不含 trait 项的 trait。它们的工作把实现类型“标记（mark）”为具有某种属性，否则就没有办法在类型系统中去表示。

```
// Impling PartialEq for a type promises
// that equality for the type has these properties:
// - symmetry: a == b implies b == a, and
// - transitivity: a == b && b == c implies a == c
// But DOES NOT promise this property:
// - reflexivity: a == a
trait PartialEq {
    fn eq(&self, other: &Self) -> bool;
}

// Eq has no trait items! The eq method is already
// declared by PartialEq, but "impling" Eq
// for a type promises this additional equality property:
// - reflexivity: a == a
trait Eq: PartialEq {}

// f64 impls PartialEq but not Eq because NaN != NaN
// i32 impls PartialEq & Eq because there's no NaNs :)
```

自动 Trait (Auto Trait)

自动 Trait 是指如果一个类型的所有成员都实现了该 trait，该类型就会自动实现该 trait。“成员 (member)”的含义取决于类型，例如：结构体的字段、枚举的变量、数组的元素、元组的项，等等。

所有的自动 trait 都是标记 trait，但不是所有的标记 trait 都是自动 trait。自动 trait 必须是标记 trait，所以编译器可以为它们提供一个自动的默认实现，如果它们有任何 trait 项，这就不可能实现了。

自动 trait 的例子。

```
// implemented for types which are safe to send between threads
unsafe auto trait Send {}

// implemented for types whose references are safe to send between threads
unsafe auto trait Sync {}
```

不安全 Trait (Unsafe Trait)

Trait 可以被标记为 unsafe，以表明实现该 trait 可能需要 unsafe 代码。**Send** 和 **Sync** 都被标记为 unsafe，因为如果它们不是自动实现的类型，就意味着它必须包含一些非 **Send** 或非 **Sync** 的成员，如果我们想手动标记类型为 **Send** 和 **Sync**，作为实现者我们必须格外小心，确保没有数据竞争。



参考资料

[1]

blanket impls: <https://doc.rust-lang.org/book/ch10-02-traits.html?highlight=blanket#using-trait-bounds-to-conditionally-implement-methods>

[2]

Sizeness in Rust: <https://github.com/pretzelhammer/rust-blog/blob/master/posts/sizedness-in-rust.md>