【译】Rust标准库Trait指南（二）

mp.weixin.qq.com/s/q_jgpeQZtsJgcI4FDfneDg

> 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）

【译】Rust标准库Trait指南（一）
【译】Rust中的Sizeness

自动 Trait

Send & Sync

所需预备知识

- 标记 Trait（Marker Trait）
- 自动 Trait （Auto Trait）
- 不安全 Trait （Unsafe Trait）

```
unsafe auto trait Send {}
unsafe auto trait Sync {}
```

如果一个类型是 `Send` ，这就意味着它可以在线程之间被安全地发送（send）。如果一个类型是 `Sync` ，这就意味着它可以在线程间安全地共享引用。说得更准确点就是，当且仅当 `&T` 是 `Send` 时，类型 `T` 是 `Sync` 。

几乎所有的类型都是 `Send` 和 `Sync` 。唯一值得注意的 `Send` 例外是 `Rc` ， `Sync` 例外中需要注意的是 `Rc` ， `Cell` ， `RefCell` 。如果我们需要一个满足 `Send` 的 `Rc` ，我们可以使用 `Arc` 。如果我们需要一个 `Cell` 或 `RefCell` 的 `Sync` 版本，我们可以使用 `Mutex` 或 `RwLock` 。尽管我们使用 `Mutex` 和 `RwLock` 来包装一个原始类型，但通常来讲，使用标准库提供的原子类型会更好一些，比如 `AtomicBool` ， `AtomicI32` ， `AtomicUsize` 等等。

几乎所有的类型都是 `Sync` 这件事，可能会让一些人感到惊讶，但它是真的，即使是对于没有任何内部同步的类型来讲，也是如此。这能够得以实现要归功于 Rust 严格的借用规则。

我们可以传递同一份数据的若干个不可变引用到多个线程中，由于只要有不可变引用存在，Rust 就会静态地保证底层数据不被修改，所以我们可以保证不会发生数据竞争。

```
use crossbeam::thread;

fn main() {
letmut greeting = String::from("Hello");
let greeting_ref = &greeting;



    thread::scope(|scoped_thread| {
// spawn 3 threads
for n in1..=3 {
// greeting_ref copied into every thread
            scoped_thread.spawn(move |_| {
println!("{} {}", greeting_ref, n); // prints "Hello {n}"
            });
        }



// line below could cause UB or data races but compiler rejects it
        greeting += " world"; // ❌ cannot mutate greeting while immutable refs exi
    });

    // can mutate greeting after every thread has joined
    greeting += " world"; // ✅
    println!("{}", greeting); // prints "Hello world"
}
```

同样地，我们可以把数据的一个可变引用传递给一个单独的线程，由于 Rust 静态地保证不存在可变引用的别名，所以底层数据不会通过另一个可变引用被修改，因此我们也可以保证不会发生数据竞争。

```
use crossbeam::thread;

fn main() {
letmut greeting = String::from("Hello");
let greeting_ref = &mut greeting;



    thread::scope(|scoped_thread| {
// greeting_ref moved into thread
        scoped_thread.spawn(move |_| {
            *greeting_ref += " world";
println!("{}", greeting_ref); // prints "Hello world"
        });



// line below could cause UB or data races but compiler rejects it
        greeting += "!!!"; // ❌ cannot mutate greeting while mutable refs exist
    });

    // can mutate greeting after the thread has joined
    greeting += "!!!"; // ✅
    println!("{}", greeting); // prints "Hello world!!!"
}
```

这就是为什么大多数类型在不需要任何显式同步的情况下，都满足 `Sync` 的原因。当我们需要在多线程中同时修改某个数据 `T` 时，除非我们用 `Arc<Mutex<T>>` 或者 `Arc<RwLock<T>>` 来包装这个数据，否则编译器是不会允许我们进行这种操作，所以编译器会在需要时强制要求进行显式地同步。

Sized

预备知识：

- 标记 Trait （Marker Trait）
- 自动 Trait （Auto Trait）

如果一个类型是 `Sized` ，这意味着它的类型大小在编译期是可知的，并且可以在栈上创建一个该类型的实例。

类型的大小及其含义是一个微妙而巨大的话题，影响到编程语言的许多方面。因为它十分重要，所以我单独写了一篇文章Sizedness in Rust[1]，如果有人想要更深入地了解sizedness，我强烈推荐阅读这篇文章。我会把这篇文章的关键内容总结在下面。

1. 所有的泛型类型都有一个隐含的 `Sized` 约束。

```
fn func<T>(t: &T) {}

// example above desugared
fn func<T: Sized>(t: &T) {}
```

2. 因为所有的泛型类型上都有一个隐含的 `Sized` 约束，如果我们想要选择退出这个约束，我们需要使用特定的“宽松约束（relaxed bound）”语法—— `?Sized` ，该语法目前只为 `Sized` trait 存在。

```
// now T can be unsized
fn func<T: ?Sized>(t: &T) {}
```

3. 所有的 trait 都有一个隐含的 `?Sized` 约束。

```
trait Trait {}

// example above desugared
trait Trait: ?Sized {}
```

这是为了让 trait 对象能够实现 trait，重申一下，所有的细枝末节都在Sizedness in Rust 中。

参考资料

[1]
Sizedness in Rust: *https://github.com/pretzelhammer/rust-blog/blob/master/posts/sizedness-in-rust.md*