腾讯课堂 零声教育 C/C++Linux服务器开发/高级架构师【零声教育】-学习视频教程-腾讯课堂 (qq.com)

1 生命周期

为什么需要生命周期：通过生命周期阻止悬空指针

生命周期为什么会存在。它们被用于什么目标？生命周期帮助编译器执行一个简单的规则：**引用不应该活得比所指对象长(no reference should outlive its referent）。**

rust的生命周期被宽泛地用来指代三种不同的东西：变量真实的生命周期、生命周期约束和生命周期标注。

1.1 生命周期的意义

1.1.1 变量的生命周期(Lifetimes of variables)

变量的生命周期是指它活着的时间。即事物存在或有效的持续时间*(the duration of a thing's existence or usefulness)*。例如，在下面的代码中，x的生命周期延续到外部块的结尾，而y的生命周期在内部块的结尾就结束了。
ex1_1_1.rs

```
{
    let x: Vec<i32> = Vec::new();//---------------------+
    {//                                                 |
        let y = String::from("why");//---+              | x's lifetime
        //                               | y's lifetime |
    }// <-------------------------------+              |
}// <-----------------------------------------------+
```

1.1.2 生命周期约束

变量在代码中的交互方式对它们的生命周期有一定的约束。例如，在下面的代码中，x=&y;这一行添加了一个约束，x的生命周期应该与封闭于y的生命周期之内。
ex1_1_2.rs

```
//error:`y` does not live long enough
{
    let x: &Vec<i32>;           // ---------+-- 'a
    {
        let y = Vec::new();//----+--- 'b  |
//                               | y's lifetime
//                               |
        x = &y;//----------------|--------------+
//                               |              |
    }// <-------------------------+              | x's lifetime
    println!("x's length is {}", x.len());//    |
}// <-----------------------------------------+
```

如果没有添加这个约束，x会在println!这行代码中访问无效的内存，因为x是y的一个引用，而y会在前面的一行被销毁。

记住，约束没有改变真实的生命周期——例如，x的生命周期，仍然延展到外部块的结尾，它们只是编译器用来禁止悬垂引用的工具。并且在上面的例子中，真实的生命周期没有满足约束：x的生命周期超出y的生命周期。因此，这段代码会编译失败。

1.1.3 生命周期标注

生命周期参数放在尖括号<>内，通过类似于`'a`来命名。

正如在上一节所见，很多次编译器都会生成所有的生命周期约束。但是随着代码愈加复杂，编译器会让程序员手动添加约束。程序员通过生命周期标注来完成这件事。例如，在下面的代码片段里，编译器需要知道print_net函数返回的引用是借用了s1还是s2。所以编译器让程序员显式地添加这个约束:
ex1_1_3_1.rs

```
//error:missing lifetime specifier
//this function's return type contains a borrowed value,
//but the signature does not say whether it is borrowed from `s1` or `s2`
fn print_ret(s1: &str, s2: &str) -> &str {
    println!("s1 is {}", s1);
    s2
}
fn main() {
    let some_str: String = "Some string".to_string();
    let other_str: String = "Other string".to_string();
    let s1 = print_ret(&some_str, &other_str);
}
```

然后程序员就对s2和返回的引用使用'a进行标注，从而告诉编译器返回值借用自s2
ex1_1_3_2.rs

```
// 注意：<'a>
// fn print_ret(s1: &str, s2: &'a str) -> &'a str { //错误的写法
fn print_ret<'a>(s1: &str, s2: &'a str) -> &'a str {
    println!("s1 is {}", s1);
    s2
}
fn main() {
    let some_str: String = "Some string".to_string();
    let other_str: String = "Other string".to_string();
    let s1 = print_ret(&some_str, &other_str);
    println!("s1:{}", s1);
}
```

生命周期标注'a在参数s2和返回的引用上都出现了，不要把这解释为s2和返回的引用有完全相同的生命周期，而应该这样解释：使用'a标注的返回的引用借用自具有相同标注的参数。
s2借用自other_str，这里的生命周期约束是，返回的引用必须不能活得比other_str长。代码可以编译是因为生命周期约束得到了满足：

```
fn print_ret<'a>(s1: &str, s2: &'a str) -> &'a str {
    println!("s1 is {}", s1);
    s2
}
fn main() {
    let some_str: String = "Some string".to_string();
    let other_str: String = "Other string".to_string();//-------------+
    let ret = print_ret(&some_str, &other_str);//---+                 |
other_str's lifetime
    //                                             | ret's lifetime  |
}// <-----------------------------------------------+-----------------+
```

1.1.4 更多示例

ex1_1_4_1.rs

```
fn min<'a>(x: &'a i32, y: &'a i32) -> &'a i32 {
    if x < y {
        x
    } else {
        y
    }
}
fn main() {
    let p = 42;
    {
        let q = 10;
        let r = min(&p, &q);
        println!("Min is {}", r);
    }
}
```

一般而言，当相同的生命周期参数标注了一个函数中两个及以上的参数，返回的引用必须不能活得比参数生命周期中最小的那个长。

许多 C++新手程序员经常会犯一个错误，**即返回一个指向局部变量的指针。**类似的尝试在 Rust 中是不被允许的
ex1_1_4_2.rs

```
//Error:cannot return reference to local variable `i`
fn get_int_ref<'a>() -> &'a i32 {
    let i: i32 = 42;
    &i
}
fn main() {
    let j = get_int_ref();
}
```

1.2 生命周期省略(Lifetime elision)

当编译器让程序员省略生命周期标注时，被称为生命周期省略(lifetime elision)。再一次地，生命周期省略被误解——生命周期与变量的产生和销毁有着密不可分的联系，怎么能被省略呢？被省略的不是生命周期，而是生命周期标注以及对应的生命周期约束。在 Rust 编译器的早期版本中，生命周期标注不允许被省略并且每个生命周期标注都是需要的。但是随着时间推移，编译器团队观察到，同样的生命周期标注不断重复，所以编译器就被修改从而开始能**推导出生命周期标注。**

在没有显示标注的情况下，编译器目前使用了3种规则来计算引用的生命周期。

第一条规则作用域输入生命周期，第二条和第三条规则作用于输出生命周期。当**编译器检查完这3条规则后仍有无法计算出生命周期的引用时，编译器就会停止运行并抛出错误。**这些规则不但对fn定义生效，也对impl代码块生效。

- 第一条规，每一个引用参数都会拥有自己的生命周期参数。
- 第二条规，当只存在一个输入生命周期参数时，这个生命周期会被赋予给所有输出生命周期参数。
- 第三条，当拥有多个输入生命周期参数，而其中一个是&self或&mut self时，self的生命周期会被赋予给所有的输出生命周期参数。

1.3 静态生命周期

'static，其生命周期**能够**存活于整个程序期间。

所有的字符串字面值都拥有 'static 生命周期，我们也可以**选择**像下面这样标注出来：
ex1_3.rs

```
fn get_str<'a>() -> &'a str {
    let s: &'static str = "I have a static lifetime.";
    s
}


fn main() {
    println!("s:{}", get_str());
}
```

为引用指定'static生命周期前要三思：**是否需要引用在程序整个生命周期内都存活。**

1.4 结合泛型类型参数、trait bounds 和生命周期

ex1_4_1.rs

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest_with_an_announcement(
        string1.as_str(),
        string2,
        "Today is someone's birthday!",
    );
    println!("The longest string is {}", result);
}

use std::fmt::Display;
```

```rust
fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

这里的T是指实现了Display trait的类型。 **生命周期属于泛型的一种，所以'a放到了<>里。**

当编译器看到`x: &'a str`的时候，`'a`会被编译器推断为x的生命周期，当编译器看到`y: &'a str`的时候，编译器会将`'a`推断为y的生命周期，但是此时有冲突，于是编译器会将`'a`推断为x和y的生命周期中最小的那个。

函数的生命周期声明是函数的入参和返回值的一种生命周期约定和限制。对于上述函数来说，生命周期参数限制了你的返回值只能是x和y的生命周期的最小的那个。

同时注意，生命周期声明类似于变量类型声明，不会改变对象的真正生命周期。当你生命的生命周期和实际不符合的时候，编译器会报错。

ex1_4_2.rs 报错

```rust
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(
        string1.as_str(),
        string2
    );
    println!("The longest string is {}", result);
}

fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

上述函数声明了`'a`和`'b`生命周期参数，分别对应x和y的生命周期。函数返回值声明只会用x的生命周期。但是编译器发现函数可能返回y，于是编译器会报错。

```
42 | fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
   |                                   -------     -------
   |                                   |
   |                                   this parameter and the return type are
declared with different lifetimes...
...
46 |         y
   |         ^ ...but data from `y` is returned here
```

编译器告诉我们返回的生命周期和声明的不符合。

1.5 Struct定义的生命周期声明

当我们定义的struct的里面有对象引用的时候，我们需要在struct的模板参数中增加生命周期声明。

ex1_5_1.rs

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
}
```

上述例子中我们定义了ImportantExcerpt struct，其中part是一个字符串引用。此时我们需要声明part的生命周期。在这里生命周期`'a`便是part的生命周期，编译器会在编译的时候检查ImportantExcerpt和`'a`，如果ImportantExcerpt大于`'a`，则会报错。

跟函数生命周期声明类似，当一个生命周期参数修饰多个字段的时候，编译器会将这个生命周期参数推断出这几个字段生命周期最小的那个。

ex1_5_2.rs 程序编译报错。

```
struct Foo<'a> {
    x: &'a i32,
    y: &'a i32,
}

fn main() {

    let x = 6;
    let m;

    {
        let y = 6;
        let f = Foo { x: &x, y: &y };  // y borrowed value does not live long
enough
        m = f.x;
    }
```

```
    println!("{}", m);
}
```

在上述例子中，Foo包含一个生命周期参数，这个生命周期参数修饰了x和y结构体字段，`'a`将会被编译器推断为x和y的生命周期中最小的那个，在这个例子中，`'a`会被推断为y的生命周期。 在这个例子中，编译器会报错，因为x的生命周期被声明为和y的生命周期一样，所以当打印m的时候，x会被编译器认为已经无效。

```
error[E0597]: `y` does not live long enough
  --> src/main.rs:13:33
   |
13 |         let f = Foo { x: &x, y: &y };
   |                                 ^^ borrowed value does not live long enough
14 |         m = f.x;
15 |     }
   |     - `y` dropped here while still borrowed
16 |
17 |     println!("{}", m);
   |                    - borrow later used here
```

上面错误告诉我们y的生命周期不够长。其实就是说r引用的生命周期应该至少要到print那一样，但是y的生命周期没有那么长，于是就报错了。

我们可以将我们的代码修改成如下：

```
struct Foo<'a, 'b> {
    x: &'a i32,
    y: &'b i32,
}

fn main() {

    let x = 6;
    let m;

    {
        let y = 6;
        let f = Foo { x: &x, y: &y };
        m = f.x;
    }

    println!("{}", m);
}
```

此时`'a`将会被推断为x的生命周期，`'b`将会被推断为y的生命周期，编译将会通过！

2 闭包

为什么需要闭包：rust中闭包本质就是一个匿名函数，它与函数最大的区别之一，在于闭包能捕获上下文环境中的变量；之二如果该匿名函数的功能只在当前函数被调用，则使用闭包更简洁。

Rust 的 **闭包**（*closures*）是可以保存进变量或作为参数传递给其他函数的匿名函数。可以在一个地方创建闭包，然后在不同的上下文中执行闭包运算。不同于函数，闭包允许捕获调用者作用域中的值。
重点：论闭包的语法、类型推断和 trait。

为什么使用闭包?

2.1 定义闭包

像定义变量一样定义闭包，想象一下RUST中的箭头函数，区别只是把包裹参数的圆括号换成了两条竖线：

```
let add = |a: u32, b: u32| -> u32 {
  a + b
};
add(1, 2); // 3
```

闭包并**不强制要求标注参数和返回值的类型**，编译器会**自行推断（**类型推断**）**：

```
let add = |a, b| {
  a + b
};
```

ex2_1.rs

```
fn main() {
    let add = |a, b| {
        a + b
    };
    add(1, 2); // 3
    add(1, 2.1); // 在第5行已经被自动推导过
}
```

如果闭包体内只有一个表达式，可以更加简写：

```
let add = |a, b| a + b;
```

函数不支持对函数体以外的变量进行引用，而**闭包中允许对外部变量进行使用**：

```
let a = 1;
let b = 2;
let add = || {
  a + b
};
add(); // 3
```

函数不支持使用外部变量：

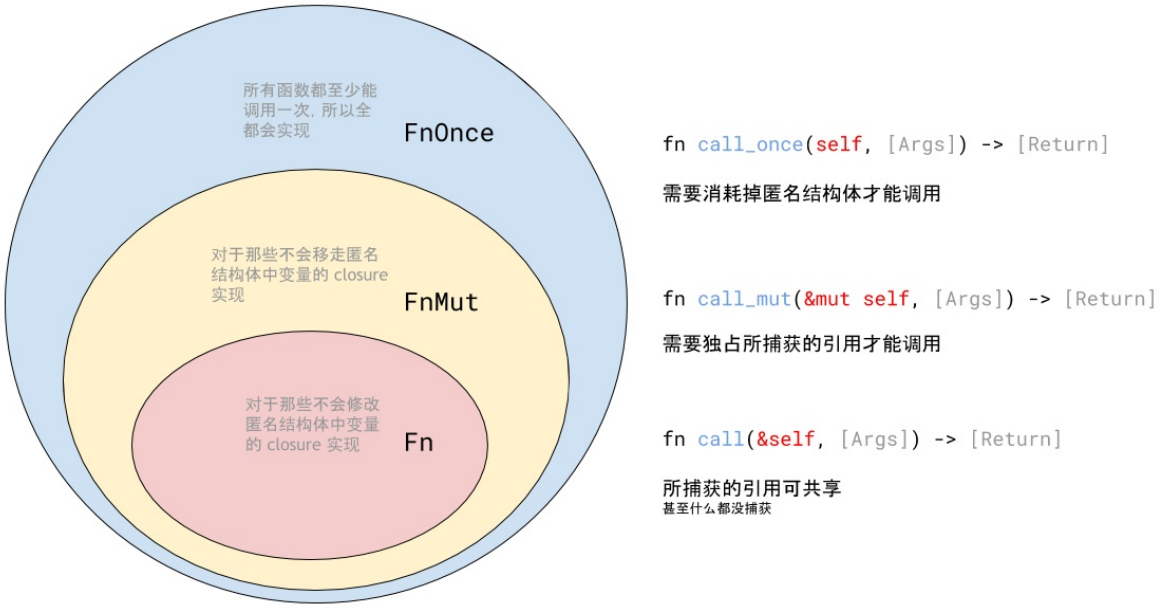
```
let a = 1;
let b = 2;
fn add() -> i32 {
  a + b // error，找不到变量a，b
};
add();
```

2.2 使用泛型和 Fn trait存储闭包

当闭包从环境中捕获一个值，闭包会在闭包体中储存这个值以供使用。这**会使用内存并产生额外的开销**，在更一般的场景中，当我们不需要闭包来捕获环境时，我们不希望产生这些开销。因为函数从未允许捕获环境，定义和使用函数也就从不会有这些额外开销。

闭包可以通过三种方式捕获其环境，他们直接对应函数的三种获取参数的方式：获取所有权，可变借用和不可变借用。这三种捕获值的方式被编码为如下三个 Fn trait：

- **Fn** trait，表示闭包以**不可变引用**的方式来捕获环境中的自由变量，同时也表示该闭包没有改变环境的能力，并且可以多次调用。对应**&self**。
- **FnMut** trait，表示闭包以**可变引用**的方式来捕获环境中的自由变量，同时意味着该闭包有改变环境的能力，也可以多次调用。对应**&mut self**。
- **FnOnce** trait，表示闭包通过**转移所有权**来捕获环境中的自由变量，同时意味着该闭包没有改变环境的能力，只能调用一次，因为该闭包会消耗自身。对应**self**。



创建闭包时，通过闭包对环境值的使用，Rust推断出具体使用哪个trait:

- 无需可变访问捕获变量的闭包实现了Fn
- 没有移动捕获变量的实现了FnMut
- 所有的闭包都实现了FnOnce

2.2.1 Fn闭包

ex2_2_1.rs

```
fn main(){
    let a =String::from("Hey !");
    let fn_closure = ||{
        println!("Closure says:{}",a);
    };
    fn_closure();
    println!("Main says:{}", a);

}
```

Fn 闭包对环境中a的捕获，是以一种类似函数中**不可变引用**传参的方式进行的，**闭包没有取得a的所有权**，所以在闭包调用完之后，在println!中还可以正常使用a。

2.2.2 FnMut闭包

ex2_2_2.rs

```
fn main(){
    let mut a =String::from("Hey !");
    let mut fn_closure = ||{
        a.push_str(" Alice");
    };
    fn_closure();
    println!("Main says:{}", a);

}
```

当闭包中会修改外部环境的自由变量值时，函数闭包fn_mut_closure也得加上mut关键词。当闭包执行时，它默认以**可变引用**的形式传入闭包，闭包没有获取其所有权，**所以闭包执行结束后，println!可以正常输出a的值。**

2.2.3 FnOnce闭包

ex2_2_3.rs

```
fn main(){
    let a = Box::new(23);
    let call_me = || {
        let c = a;
    };
    println!("a:{}", a); // 如果加上这一行
    call_me();
    call_me();
}
```

call_me 第一次执行完之后，之前捕获的变量已经被释放了，所以已经无法在执行第二次了。所以，如果要运行多次，可以使用 FnMut\Fn。

2.2.4 mov闭包强行获得环境中自由变量的所有权

如果希望强制闭包获取其使用的环境值的所有权，可以**在参数列表前使用 move 关键字。**这个技巧在将闭包传递给新线程以便将数据移动到新线程中时最为实用。

ex2_2_4.rs

```
fn main(){
    let a =String::from("Hey !");
    let fn_closure = move ||{
        println!("Closure says:{}",a);
    };
    fn_closure();
    println!("Main says:{}", a);

}
```

闭包获得了a的所有权，因此编译会出错

3 智能指针

指针 （*pointer*）是一个包含内存地址的变量的通用概念。这个地址引用，或 “指向”（points at）一些其他数据。
Rust 中最常见的指针是前面课程讲的 **引用**（*reference*）。引用以 & 符号为标志并借用了他们所指向的值。
智能指针（*smart pointers*）是一类数据结构，他们的表现类似指针，但是也拥有额外的元数据和功能。智能指针的概念并不为 Rust 所独有；其起源于 C++ 并存在于其他语言中。Rust 标准库中不同的智能指针提供了多于引用的额外功能。
引用计数 （*reference counting*）智能指针类型，其允许数据有多个所有者。引用计数智能指针记录总共有多少个所有者，并当没有任何所有者时负责清理数据。

在 Rust 中，普通引用和智能指针的一个额外的区别是引用是一类只借用数据的指针；相反，在大部分情况下，智能指针 **拥有** 他们指向的数据。

考虑到智能指针是一个在 Rust 经常被使用的通用设计模式，本课程并不会覆盖所有现存的智能指针。很多库都有自己的智能指针而你也可以编写属于你自己的智能指针。这里将会讲到的是来自标准库中最常用的一些：

- Box，用于在堆上分配值
- Rc，一个引用计数类型，其数据可以有多个所有者
- Ref 和 RefMut，通过 RefCell 访问。（ RefCell 是一个在运行时而不是在编译时执行借用规则的类型）。

3.1 Box

装箱类型（box）使我们可以将数据存储在堆上，并在栈中保留一个指向堆数据的指针。

3.1.1 使用场景

- 当拥有一个无法在编译时确定大小的类型，但又想要在一个要求固定尺寸的上下文环境中使用这个类型的值时。
- 当需要传递大量数据的所有权，但又不希望产生大量数据的复制行为时。
- 当希望拥有一个实现了指定trait的类型值，但又不关心具体的类型时。

ex3_1_1.rs

```
fn main(){
    let b = Box::new(5);
    println!("{}", b); // 5
}
```

另外，和其他任何拥有所有权的值一样，装箱会在离开自己的作用域时被释放,包括指针和堆上的数据。

3.1.2 使用装箱定义递归类型

rust中使用基本类型是无法实现递归的，当然你也可以自行试一下实现一个链表数据结构，过程中必然会得到rust的报错，因为rust无法确定递归类型的大小，进而无法通过编译。

3.1.3 使用枚举实现链表结构

下面我们使用rust尝试实现链表，看一下rust的报错：
ex3_1_3.rs

```
// 该程序报错
#[derive(Debug)]
enum List {
  Cons(i32, List), // error, 递归类型有无限的大小
  Nil,
}
fn main() {
    let list = List::Cons(1,
        List::Cons(2,
            List::Cons(3, List::Nil)
        )
    );
    println!("{:?}", list);
}
```

由于rust在编译过程中需要知道所有类型的大小，而在运行时递归是无限的，所以在编译过程中无法计算出递归的大小

3.1.4 使用Box将递归类型的大小固定下来

下面使用Box类型重新尝试看看如何实现链表：
ex3_1_4ok.rs

```
#[derive(Debug)]
enum List {
  Node(i32, Box<List>), // 使用Box标记类型
  Nil,
}
fn main() {
    let list = List::Node(1,
    Box::new(List::Node(2,
        Box::new(List::Node(3,
            Box::new(List::Nil))
        )
    ))
    );
    println!("{:?}", list);
}
```

因为**Box是一个指针**，所以Rust可以在编译时就确定一个Box的具体大小。指针的大小总是恒定的，它不会因为指向数据的大小而产生变化。

3.1.5 结构体实现链表结构

ex3_1_5list.rs

```
//复习
// pub enum Option<T> {
//     None,
```

```
//     Some(T),
// }
// Option主要有以下一些用法：
// 初始化值；
// 作为在整个输入范围内没有定义的函数的返回值；
// 作为返回值，用None表示出现的简单错误；
// 作为结构体的可选字段；
// 作为结构体中可借出或者是可载入的字段；
// 作为函数的可选参数；
// 代表空指针；
// 用作复杂情况的返回值。

// 定义节点
#[derive(Debug)]
struct Node {
    val: i32,
    next: Option<Box<Node>>, // 节点的next指向下一个节点，可能为空，故类型为Option； 避免编译器无法计算节点大小，用Box包裹Node；
}

impl Node {
    fn new(val: i32) -> Box<Node> {
        Box::new(Node { val, next: None })
    }

    fn link(&mut self, node: Box<Node>) {
        self.next = Some(node);
    }
}
// 定义链表结构体
#[derive(Debug)]
struct BoxedLinkedList {
    head: Option<Box<Node>>,
}

impl BoxedLinkedList {
    fn new() -> BoxedLinkedList {
        BoxedLinkedList { head: None }
    }

    fn display(&self) {
        println!("{:?}", self);
    }
    fn is_empty(&self) -> bool {
        self.head.is_none()
    }
    /*
    将list.head 这个Option包含的值用前面学到的take()方法取出；
    然后link到新节点后面；
    头指针指向新的节点；
    */
    fn push_front(&mut self, val: i32) {
        let mut new_node = Node::new(val);
        if let Some(node) = self.head.take() {
            new_node.link(node);
        }
        self.head = Some(new_node);
    }
```

```
    fn pop_front(&mut self) -> Option<i32> {
        // 获取头指针的包裹的值；生成一个新的Option；
        match self.head.take() {
            None => None, // 为空的话直接返回None；
            Some(node) => {
                // 注意此时拥有 node 的所有权；
                self.head = node.next; // Box自动解引用，然后将node的next的所有权转移
给变量head；
                Some(node.val)
            }
        }
    }
}

fn main() {
    let mut list = BoxedLinkedList::new();
    for i in 1..3 {
        list.push_front(i);
    }

    list.display();

    println!();

    while !list.is_empty() {
        let v = list.pop_front().unwrap(); // unwrap 正常是返回对应的正常值，异常是直
接panic
        println!("pop '{}' from head of list", v);
        print!("now list is : ");
        list.display();
    }
}
```

3.2 Rc 与 Arc

Rc 主要用于同一堆上所分配的数据区域需要有多个只读访问的情况，比起使用 Box 然后创建多个不可变引用的方法更优雅也更直观一些，以及比起单一所有权，**Rc 支持多所有权。**
Rc 为 Reference Counter 的缩写，即为引用计数，Rust 的 Runtime 会实时记录一个 Rc 当前被引用的次数，并在引用计数归零时对数据进行释放（类似 Python 的 GC 机制）。因为需要维护一个记录 Rc 类型被引用的次数，所以这个实现需要 Runtime Cost。
ex3_2_1.rs

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(1);
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Rc::clone(&a);
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Rc::clone(&a);
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

输出依次会是 1 2 3 2。
需要注意的主要有两点。

- 首先， Rc 是**完全不可变的**，可以将其理解为对同一内存上的数据同时存在的多个只读指针。
- 其次，Rc 是只适用于**单线程内的**，尽管从概念上讲不同线程间的只读指针是完全安全的，但由于 Rc 没有实现在多个线程间保证计数一致性，所以如果你尝试在多个线程内使用它，会得到这样的错误：

```
use std::thread;
use std::rc::Rc;

fn main() {
    let a = Rc::new(1);
    thread::spawn(|| {
        let b = Rc::clone(&a);
        // Error: `std::rc::Rc<i32>` cannot be shared between threads safely
    }).join();
}
```

如果想在不同线程中使用 Rc 的特性该怎么办呢？答案是** Arc**，即 Atomic reference counter。此时引用计数就可以在不同线程中安全的被使用了。
ex3_2_2.rs

```
use std::thread;
use std::sync::Arc;

fn main() {
    let a = Arc::new(1);
    thread::spawn(move || {
        let b = Arc::clone(&a);
        println!("{}", b);  // Output: 1
    }).join();
}
```

3.3 Cell

Cell 其实和 Box 很像，但后者同时不允许存在多个对其的可变引用，如果我们真的很想做这样的操作，在需要的时候随时改变其内部的数据，而不去考虑 Rust 中的不可变引用约束，就可以使用 Cell。Cell 允许多个共享引用对其内部值进行更改，实现了「内部可变性」。

ex3_3_1.rs

```
use std::cell::Cell;
fn main() {
    let x = Cell::new(1);
    let y = &x;
    let z = &x;
    x.set(2);
    y.set(3);
    z.set(4);
    println!("{}", x.get());  // Output: 4
}
```

这段看起来非常不 Rust 的 Rust 代码其实是可以通过编译并运行成功的，Cell 的存在看起来似乎打破了 Rust 的设计哲学，但由于仅仅对实现了 Copy 的 T，Cell 才能进行 .get() 和 .set() 操作。而实现了 Copy 的类型在 Rust 中几乎等同于会分配在栈上的数据（可以直接按比特进行连续 n 个长度的复制），所以对其随意进行改写是十分安全的，不会存在堆数据泄露的风险（比如我们不能直接复制一段栈上的指针，因为指针指向的内容可能早已物是人非）。也是得益于 Cell 实现了外部不可变时的内部可变形，可以允许以下行为的发生：
ex3_3_2.rs

```
use std::cell::Cell;

fn main() {
    let a = Cell::new(1);

    {
        let b = &a;
        b.set(2);
    }


    println!("{:?}", a);  // Output: 2
}
```

如果换做 Box，则在中间出现的 Scope 就会使 a 的所有权被移交，且在执行完毕之后被 Drop。最后还有一点，Cell 只能在单线程的情况下使用。

3.4 RefCell

因为 Cell 对 T 的限制：只能作用于实现了 Copy 的类型，所以应用场景依旧有限（安全的代价）。但是我如果就是想让任何一个 T 都可以塞进去该咋整呢？RefCell 去掉了对 T 的限制，但是别忘了要牢记初心，不忘继续践行 Rust 的内存安全的使命，既然不能在读写数据时简单的 Copy 出来进去了，该咋保证内存安全呢？相对于标准情况的静态借用，RefCell 实现了运行时借用，这个借用是临时的，而且 Rust 的 Runtime 也会**随时紧盯 RefCell 的借用行为**：同时只能有一个可变借用存在，否则直接 Painc。也就是说 RefCell 不会像常规时一样在编译阶段检查引用借用的安全性，而是在程序运行时动态的检查，**从而提供在不安全的行为下出现一定的安全场景的可行性。**
ex3_4.rs

```
use std::cell::RefCell;
use std::thread;

fn main() {
    thread::spawn(move || {
       let c = RefCell::new(5);
       let m = c.borrow();

       let b = c.borrow_mut(); // thread '<unnamed>' panicked at 'already
borrowed: BorrowMutError', ex3_3_3.rs:9:18
    }).join();
}
```

如上程序所示，如同一个读写锁应该存在的情景一样，**直接进行读后写是不安全的**，所以 borrow 过后 borrow_mut 会导致程序 Panic。同样，ReCell 也只能在单线程中使用。
如果你要实现的代码很难满足 Rust 的编译检查，不妨考虑使用 Cell 或 RefCell，它们在最大程度上以安全的方式给了你些许自由，但别忘了时刻警醒自己自由的代价是什么，也许获得喘息的下一秒，一个可怕的 Panic 就来到了你身边！

3.5 组合使用

如果遇到**要实现一个同时存在多个不同所有者，但每个所有者又可以随时修改其内容**，且这个内容类型 T 没有实现 Copy 的情况该怎么办？使用 Rc 可以满足第一个要求，但是由于其是不可变的，要修改内容并不可能；使用 Cell 直接死在了 T 没有实现 Copy 上；使用 RefCell 由于无法满足多个不同所有者的存在，也无法实施。可以看到各个智能指针可以解决其中一个问题，既然如此，为何我们不把 Rc 与 RefCell 组合起来使用呢?
ex

```
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let shared_vec: Rc<RefCell<_>> = Rc::new(RefCell::new(Vec::new()));
    // Output: []
    println!("{:?}", shared_vec.borrow());
    {
        let b = Rc::clone(&shared_vec);
        b.borrow_mut().push(1);
        b.borrow_mut().push(2);
    }
    shared_vec.borrow_mut().push(3);
    // Output: [1, 2, 3]
    println!("{:?}", shared_vec.borrow());
}
```

通过 Rc 保证了多所有权，而通过 RefCell 则保证了内部数据的可变性。

3.6 Drop

`Drop` trait 只有一个方法：`drop`，当一个对象离开作用域时会自动调用该方法。`Drop` trait 的主要作用是释放实现者实例拥有的资源。

`Box`，`Vec`，`String`，`File`，以及 `Process` 是一些实现了 `Drop` trait 来释放资源的类型的例子。`Drop` trait 也可以针对任意自定义数据类型手动实现。

下面示例给 `drop` 函数增加了打印到控制台的功能，用于宣布它在什么时候被调用。

```
struct Droppable {
    name: &'static str,
}

// 这个简单的 `drop` 实现添加了打印到控制台的功能。
impl Drop for Droppable {
    fn drop(&mut self) {
        println!("> Dropping {}", self.name);
    }
}

fn main() {
    let _a = Droppable { name: "a" };

    // 代码块 A
    {
```

```
        let _b = Droppable { name: "b" };

        // 代码块 B
        {
            let _c = Droppable { name: "c" };
            let _d = Droppable { name: "d" };

            println!("Exiting block B");
        }
        println!("Just exited block B");

        println!("Exiting block A");
    }
    println!("Just exited block A");

    // 变量可以手动使用 `drop` 函数来销毁。
    drop(_a);
    // 试一试 ^ 将此行注释掉。

    println!("end of the main function");

    // `_a` **不会**在这里再次销毁，因为它已经被（手动）销毁。
}
```

4 面向对象

面向对象编程（Object-Oriented Programming，OOP）是一种模式化编程方式。面向对象编程语言所共享的一些特性主要是：

1. 对象
2. 封装
3. 继承

4.1 面向对象基础

4.1.1 对象包含数据和行为

面向对象的程序是由对象组成的。一个 **对象** 包含数据和操作这些数据的过程。这些过程通常被称为 **方法** 或 **操作**。
在这个定义下，Rust 是面向对象的：结构体和枚举包含数据而 impl 块提供了在结构体和枚举之上的方法。虽然带有方法的结构体和枚举并不被 **称为** 对象，但是他们提供了与对象相同的功能。

4.1.2 封装隐藏了实现细节

封装（*encapsulation*）思想：对象的实现细节不能被使用对象的代码获取到。所以唯一与对象交互的方式是通过对象提供的公有 API；使用对象的代码无法深入到对象内部并直接改变数据或者行为。封装使得改变和重构对象的内部时无需改变使用对象的代码。
可以使用 pub 关键字来决定模块、类型、函数和方法是公有的，而默认情况下其他一切都是私有的。
ex4_1_2

```
second.rs
pub struct ClassName {
    field: i32,
```

```
}

impl ClassName {
    pub fn new(value: i32) -> ClassName {
        ClassName {
            field: value
        }
    }

    pub fn public_method(&self) {
        println!("from public method");
        self.private_method();
    }

    fn private_method(&self) {
        println!("from private method");
    }
}
main.rs
mod second;
use second::ClassName;

fn main() {
    let object = ClassName::new(1024);
    object.public_method();
}
```

4.1.3 继承，作为类型系统与代码共享

继承（*Inheritance*）是一个很多编程语言都提供的机制，一个对象可以定义为继承另一个对象的定义，这使其可以获得父对象的数据和行为，而无需重新定义。
如果一个语言必须有继承才能被称为面向对象语言的话，那么 Rust 就不是面向对象的。无法定义一个结构体继承父结构体的成员和方法。然而，如果你过去常常在你的编程工具箱使用继承，根据你最初考虑继承的原因，Rust 也提供了其他的解决方案。
选择继承有两个主要的原因：

- 第一个是为了重用代码：一旦为一个类型实现了特定行为，继承可以对一个不同的类型重用这个实现。相反 Rust 代码可以使用默认 trait 方法实现来进行共享。
- 第二个使用继承的原因与类型系统有关：表现为子类型可以用于父类型被使用的地方。这也被称为**多态**（*polymorphism*），这意味着如果多种对象共享特定的属性，则可以相互替代使用

ex4_1_3

```
// 定义trait及方法
trait Bird {
    fn fly(&self);
}

struct Duck;
struct Swan;

// 将trait impl到 Duck中，将重写的trait方法存入虚函数表
impl Bird for Duck {
    fn fly(&self){
        println!("duck duck");
    }
```

```
}

// 将trait impl到 Swan中，将重写的trait方法存入虚函数表
impl Bird for Swan {
    fn fly(&self){
        println!("Swan Swan");
    }
}

// 定义一个调用函数
fn print_trait_obj(p: &dyn Bird){
    p.fly();
}

fn main() {
    // 新建对象
    let duck = Duck;

    // 创建 对象的引用
    let p_duck = &duck;

    // 将对象引用 转换成 trait对象，这个过程中--trait对象为胖指针(指针1.p_duck：指针2.(虚函数表+Duck的trait方法在虚函数表中的偏移量))
    let p_bird = p_duck as &dyn Bird;

    // 当调用trait方法时，从指针1中获取对象，从指针2中获取trait方法
    // print_trait_obj(p_bird);
    p_bird.fly();  // 因为fly(&self)，所以等价于 (p_bird.vtable.fly)(p_duck)

    // 同理
    let swan = Swan;
    let p_swan = &swan;
    let p_bird = p_swan as &dyn Bird;  // 指针p_bird发生了重绑定
    p_bird.fly();
    swan.fly();
}
```

5 并发编程

Rust 团队认为确保内存安全和防止并发问题是两个分别需要不同方法应对的挑战。随着时间的推移，团队发现所有权和类型系统是一系列解决内存安全 **和** 并发问题的强有力的工具！通过利用所有权和类型检查，在 Rust 中很多并发错误都是 **编译时** 错误，而非运行时错误。
重点：

- 如何创建线程来同时运行多段代码。
- **消息传递**（*Message passing*）并发，其中通道（channel）被用来在线程间传递消息。
- **共享状态**（*Shared state*）并发，其中多个线程可以访问同一片数据。
- Sync 和 Send trait，将 Rust 的并发保证扩展到用户定义的以及标准库提供的类型中

5.1 线程

多线程可能遇到的问题：

- 竞争状态（Race conditions），多个线程以不一致的顺序访问数据或资源
- 死锁（Deadlocks），两个线程相互等待对方停止使用其所拥有的资源，这会阻止它们继续运行
- 只会发生在特定情况且难以稳定重现和修复的 bug

5.1.1 使用spawn创建新线程

ex5_1_1.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
//    thread::sleep(Duration::from_millis(1000));
}
```

5.1.2 使用join等待所有线程结束

ex5_1_2.rs

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```

从 thread::spawn 保存一个 JoinHandle 以确保该线程能够运行至结束

5.1.3 线程与move闭包

move 闭包， 其经常与 thread::spawn 一起使用，因为它允许我们在一个线程中使用另一个线程的数据。

ex5_1_3fail.rs 会报错

```rust
use std::thread;

fn main() {
   let v = vec![1, 2, 3];

   let handle = thread::spawn(|| {
       println!("Here's a vector: {:?}", v);
  });

   handle.join().unwrap();
}
```

使用 move 关键字强制获取它使用的值的所有权
ex5_1_3ok.rs

```rust
use std::thread;

fn main() {
   let v = vec![1, 2, 3];

   let handle = thread::spawn(move || {
       println!("Here's a vector: {:?}", v);
  });

   handle.join().unwrap();
}
```

5.2 消息传递

思想：不要通过共享内存来通讯；而是通过通讯来共享内存。

Rust 中一个实现消息传递并发的主要工具是 **通道**（*channel*），Rust 标准库提供了其实现的编程概念。你可以将其想象为一个水流的通道，比如河流或小溪。如果你将诸如橡皮鸭或小船之类的东西放入其中，它们会顺流而下到达下游。
编程中的通道有两部分组成，一个发送者（transmitter）和一个接收者（receiver）。发送者位于上游位置，在这里可以将橡皮鸭放入河中，接收者则位于下游，橡皮鸭最终会漂流至此。代码中的一部分调用发送者的方法以及希望发送的数据，另一部分则检查接收端收到的消息。**当发送者或接收者任一被丢弃时**可以认为通道被 **关闭**（*closed*）了。

5.2.1 创建1:1的生产者消费者

ex5_2_1.rs

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

fn main() {
   let (tx, rx) = mpsc::channel();

   thread::spawn(move || {
       let val = String::from("hi");
       tx.send(val).unwrap();
  });

   let received = rx.recv().unwrap();
```

```
    println!("Got: {}", received);
}
```

recv，它是 *receive* 的缩写。这个方法会阻塞主线程执行直到从通道中接收一个值。一旦发送了一个值，recv 会在一个 Result<T, E> 中返回它。当通道发送端关闭，recv 会返回一个错误表明不会再有新的值到来了。
try_recv 不会阻塞，相反它立刻返回一个 Result<T, E>：Ok 值包含可用的信息，而 Err 值代表此时没有任何消息。如果线程在等待消息过程中还有其他工作时使用 try_recv 很有用：可以编写一个循环来频繁调用 try_recv，在有可用消息时进行处理，其余时候则处理一会其他工作直到再次检查。

5.2.2 通过克隆发送者来创建多个生产者

ex5_2_2.rs

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    let tx1 = tx.clone();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

5.3 共享状态并发

5.3.1 Mutex互斥器

互斥器（*mutex*）是 *mutual exclusion* 的缩写，也就是说，任意时刻，其只允许一个线程访问某些数据。为了访问互斥器中的数据，线程首先需要通过获取互斥器的 **锁**（*lock*）来表明其希望访问数据。锁是一个作为互斥器一部分的数据结构，它记录谁有数据的排他访问权。因此，我们描述互斥器为通过锁系统 **保护**（*guarding*）其数据。

ex5_3_1.rs 创建一个Mutex

```
use std::sync::Mutex;

fn main() {
   let m = Mutex::new(5);

  {
       let mut num = m.lock().unwrap();
       *num = 6;
  }

   println!("m = {:?}", m);
}
```

5.3.2 在线程间共享Mutex

示例：程序启动了 10 个线程，每个线程都通过 Mutex 来增加计数器的值
ex5_3_2.rs 程序会报错

```
use std::sync::Mutex;
use std::thread;

fn main() {
   let counter = Mutex::new(0);
   let mut handles = vec![];

   for _ in 0..10 {
       let handle = thread::spawn(move || {
           let mut num = counter.lock().unwrap();

           *num += 1;
       });
       handles.push(handle);
  }

   for handle in handles {
       handle.join().unwrap();
  }

   println!("Result: {}", *counter.lock().unwrap());
}
```

```
lqf@ubuntu:/mnt/hgfs/0voice/vip/rust/rust-3-src$ rustc ex5_3_2.rs
error[E0382]: use of moved value: `counter`
  --> ex5_3_2.rs:9:35
   |
5  |     let counter = Mutex::new(0);
   |         ------- move occurs because `counter` has type `Mutex<i32>`, which does not implement the `Copy` trait
...
9  |         let handle = thread::spawn(move || {
   |                                    ^^^^^^^ value moved into closure here, in previous iteration of loop
10 |             let mut num = counter.lock().unwrap();
   |                           ------- use occurs due to use in closure

error: aborting due to previous error
```

错误信息表明 counter 值在上一次循环中被移动了。所以 Rust 告诉我们不能将 counter 锁的所有权移动到多个线程中。

5.3.3 多线程和所有权

将上小节中的 Mutex 封装进 Rc 中并在将所有权移入线程之前克隆了 Rc。现在我们理解了所发生的错误，同时也将代码改回使用 for 循环，并保留闭包的 move 关键字：
ex5_3_3.rs 尝试使用 Rc 来允许多个线程拥有 Mutex **（仍然报错）**

```
use std::rc::Rc;
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Rc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Rc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

报错

```
lqf@ubuntu:/mnt/hgfs/0voice/vip/rust/rust-3-src$ rustc ex5_3_3.rs
error[E0277]: `Rc<Mutex<i32>>` cannot be sent between threads safely
   --> ex5_3_3.rs:11:21
    |
11  |         let handle = thread::spawn(move || {
    |                      ^^^^^^^^^^^^^ -
    |  ____________________|
    | |                    `Rc<Mutex<i32>>` cannot be sent between threads safely
12  | |             let mut num = counter.lock().unwrap();
13  | |
14  | |             *num += 1;
15  | |         });
    | |_______- within this `[closure@ex5_3_3.rs:11:35: 15:8]`
    |
    = help: within `[closure@ex5_3_3.rs:11:35: 15:8]`, the trait `Send` is not implemented for `Rc<Mutex<i32>>`
    = note: required because it appears within the type `[closure@ex5_3_3.rs:11:35: 15:8]`
note: required by a bound in `spawn`
```

不幸的是，Rc 并不能安全的在线程间共享。当 Rc 管理引用计数时，它必须在每一个 clone 调用时增加计数，并在每一个克隆被丢弃时减少计数。Rc 并没有使用任何并发原语，来确保改变计数的操作不会被其他线程打断。在计数出错时可能会导致诡异的 bug，比如可能会造成内存泄漏，或在使用结束之前就丢弃一个值。我们所需要的是一个完全类似 Rc，**又以一种线程安全的方式改变引用计数的类型。**

5.3.4 原子引用计数Arc

Arc**是** 一个类似 Rc 并可以安全的用于并发环境的类型。字母 “A” 代表 **原子性**（*atomic*），所以这是一个 **原子引用计数**（*Atomically reference counted*）类型。原子性是另一类这里还未涉及到的并发原语：请查看标准库中 std::sync::atomic 的文档来获取更多细节。其中的要点就是：原子性类型工作起来类似原始类型，不过可以安全的在线程间共享。
疑问：什么不是所有的原始类型都是原子性的？为什么不是所有标准库中的类型都默认使用 Arc 实现？
原因在于**线程安全带有性能消耗**，我们希望只在必要时才为此买单。如果只是在单线程中对值进行操作，原子性提供的保证并无必要，代码可以因此运行的更快。

Arc 和 Rc 有着相同的 API，所以修改程序中的 use 行和 new 调用。

ex5_3_4.rs 使用 Arc 包装一个 Mutex 能够实现在多线程之间共享所有权

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

5.4 使用Sync 和Send trait 的可扩展并发

我们之前讨论的几乎所有内容，都属于标准库，而不是语言本身的内容。由于不需要语言提供并发相关的基础设施，并发方案不受标准库或语言所限：我们可以编写自己的或使用别人编写的并发功能。

但也有两个并发概念是内嵌于语言中的：std::marker 中的 Sync 和 Send trait。

1. **通过 Send 允许在线程间转移所有权**

Send 标记 trait 表明类型的所有权可以在线程间传递。几乎所有的 Rust 类型都是Send 的，不过有一些例外，包括 Rc：这是不能 Send 的，因为如果克隆了 Rc 的值并尝试将克隆的所有权转移到另一个线程，这两个线程都可能同时更新引用计数。为此，**Rc 被实现为用于单线程场景**，这时不需要为拥有线程安全的引用计数而付出性能代价。

任何完全由 Send 的类型组成的类型也会自动被标记为 Send。几乎所有基本类型都是 Send 的，除了将会讨论的裸指针（raw pointer）。

2. **Sync 允许多线程访问**

Sync 标记 trait 表明一个实现了 Sync 的类型可以安全的在多个线程中拥有其值的引用。换一种方式来说，对于任意类型 T，如果 &T（T 的引用）是 Send 的话 T 就是 Sync 的，这意味着其引用就可以安全的发送到另一个线程。类似于 Send 的情况，**基本类型是 Sync 的**，完全由 Sync 的类型组成的类型也是 Sync 的。
智能指针 Rc 也不是 Sync 的，出于其不是 Send 相同的原因。RefCell和 Cell 系列类型不是 Sync 的。RefCell 在运行时所进行的借用检查也不是线程安全的。Mutex 是 Sync 的，正如 “在线程间共享 Mutex” 部分所讲的它可以被用来在多线程中共享访问。

3. **手动实现 Send 和 Sync 是不安全的**

通常并不需要手动实现 Send 和 Sync trait，因为由 Send 和 Sync 的类型组成的类型，自动就是 Send 和 Sync 的。因为他们是**标记 trait**，甚至都不需要实现任何方法。他们只是用来加强并发相关的不可变性的。

5.5 线程池

供参考

lib.rs

```rust
use std::sync::{mpsc, Arc, Mutex};

type Job = Box<dyn FnOnce() + Send + 'static>;

enum Message {
    Task(Job),
    Terminate,
}

// all workers get task from it
type SharedTaskReceiver = Arc<Mutex<mpsc::Receiver<Message>>>;

struct Worker {
    id: usize,
    // task return type is empty '()'
    // when drop, some threads which joined first will not own JoinHandle
    handle: Option<std::thread::JoinHandle<()>>,
}

impl Worker {
    pub fn new(id: usize, task_receiver: SharedTaskReceiver) -> Worker {
        let handle = std::thread::spawn(move || loop {
            let message = task_receiver.lock().unwrap().recv().unwrap();
            match message {
                Message::Task(job) => {
                    println!("Worker {} got a job, executing...", id);
                    job();
                }
                Message::Terminate => {
                    println!("Worker {} was told to terminate.", id);
                    break;
                }
            }
```

```
        });
        Worker {
            id: id,
            handle: Some(handle),
        }
    }
}

pub struct ThreadPool {
    workers: Vec<Worker>,
    task_sender: mpsc::Sender<Message>,
}

impl ThreadPool {
    pub fn new(size: usize) -> Result<ThreadPool, &'static str> {
        if size == 0 {
            return Err("A thread pool with 0 thread is not allowed");
        }

        let (sender, receiver) = mpsc::channel();
        let shared_receiver = Arc::new(Mutex::new(receiver));

        let mut workers = vec![];
        for i in 0..size {
            workers.push(Worker::new(i, Arc::clone(&shared_receiver)));
        }

        Ok(ThreadPool {
            workers: workers,
            task_sender: sender,
        })
    }

    pub fn execute<F: FnOnce() + Send + 'static>(&self, func: F) {
        let new_job = Message::Task(Box::new(func));
        self.task_sender.send(new_job).unwrap();
    }
}
// 为 ThreadPool 实现 Drop trait 来 join 所有的线程，保证队列中的任务完成。
impl Drop for ThreadPool {
    fn drop(&mut self) {
        println!(
            "Sending terminate message to {} workers",
            self.workers.len()
        );
        for _ in self.workers.iter() {
            self.task_sender.send(Message::Terminate).unwrap();
        }

        for worker in self.workers.iter_mut() {
            println!("Shutting down worker {}", worker.id);
            if let Some(handle) = worker.handle.take() {
                handle.join().unwrap();
            }
        }
    }
}
```

main.rs

```rust
use std::time::Duration;
use std::sync::{Arc, Mutex};
mod lib;


fn main() {
    let mut tasks = vec![];

    let counter = Arc::new(Mutex::new(0));
    let total = 200;
    for i in 0..total {
        let counter1 = Arc::clone(&counter);
        tasks.push(move || {
            std::thread::sleep(Duration::from_millis(i));
            let mut num = counter1.lock().unwrap();
            *num += 1;
            println!("num:{}", num);
        });
    }

    let pool = lib::ThreadPool::new(10).unwrap();   // 创建线程池

    for task in tasks {
        pool.execute(task);  //入队列
    }

    // guarantee to call ThreadPool::drop before check
    std::mem::drop(pool);
}
```