Rust 语法课三

河南科技大学-徐堃元

2024/4/17

本节课程内容

- 闭包与迭代器
- 智能指针
- Rust 并发编程
- Rust 异步编程简介

《Rust 程序设计语言》：

https://kaisery.github.io/trpl-zh-cn/

《通过例子学 Rust》：

https://rustwiki.org/zh-CN/rust-by-example/

Rust 语言中文社区：

https://rustcc.cn/

闭包

Rust 的 **闭包**（*closures*）是可以保存在一个变量中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，闭包允许捕获被定义时所在作用域中的值。

其本质是拥有可能关联上下文的匿名函数体

闭包类型推断和注解

```
let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };
```

为闭包的参数和返回值增加可选的类型注解

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

捕获环境

闭包可以通过三种方式捕获其环境，它们直接对应到函数获取参数的三种方式：不可变借用，可变借用和获取所有权。闭包会根据函数体中如何使用被捕获的值决定用哪种方式捕获。

```
fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let only_borrows = || println!("From closure: {:?}", list);

    println!("Before calling closure: {:?}", list);
    only_borrows();
    println!("After calling closure: {:?}", list);
}
```

捕获不可变引用。

```
Before defining closure: [1, 2, 3]
Before calling closure: [1, 2, 3]
From closure: [1, 2, 3]
After calling closure: [1, 2, 3]
```

捕获可变引用


```rust
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    let mut borrows_mutably = || list.push(7);

    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

使用 move 强制获取所有权

在将闭包传递到一个新的线程时这个技巧很有用，它可以移动数据所有权给新线程。

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

Fn trait

闭包捕获和处理环境中的值的方式影响闭包实现的 trait。Trait 是函数和结构体指定它们能用的闭包的类型的方式。取决于闭包体如何处理值，闭包自动、渐进地实现一个、两个或三个 `Fn` trait。

1. `FnOnce` 适用于能被调用一次的闭包，所有闭包都至少实现了这个 trait，因为所有闭包都能被调用。一个会将捕获的值移出闭包体的闭包只实现 `FnOnce` trait，这是因为它只能被调用一次。
2. `FnMut` 适用于不会将捕获的值移出闭包体的闭包，但它可能会修改被捕获的值。这类闭包可以被调用多次。
3. `Fn` 适用于既不将被捕获的值移出闭包体也不修改被捕获的值的闭包，当然也包括不从环境中捕获值的闭包。这类闭包可以被调用多次而不改变它们的环境，这在会多次并发调用闭包的场景中十分重要。

迭代器

迭代器模式允许你对一个序列的项进行某些处理。**迭代器**（*iterator*）负责遍历序列中的每一项和决定序列何时结束的逻辑。当使用迭代器时，我们无需重新实现这些逻辑。

在 Rust 中，迭代器是 **惰性的**（*lazy*），这意味着在调用方法使用迭代器之前它都不会有效果。

```
let v1 = vec![1, 2, 3];

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

Iterator trait 和 next 方法

迭代器都实现了一个叫做 `Iterator` 的定义于标准库的 trait。这个 trait 的定义看起来像这样：

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // 此处省略了方法的默认实现
}
```

`next` 是 `Iterator` 实现者被要求定义的唯一方法。`next` 一次返回迭代器中的一个项，封装在 `Some` 中，当迭代器结束时，它返回 `None`。

可以直接调用迭代器的 `next` 方法

```
    #[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];
        let mut v1_iter = v1.iter();
        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
```

消费迭代器的方法

`Iterator` trait 有一系列不同的由标准库提供默认实现的方法；你可以在 `Iterator` trait 的标准库 API 文档中找到所有这些方法。一些方法在其定义中调用了 `next` 方法，这也就是为什么在实现 `Iterator` trait 时要求实现 `next` 方法的原因。

这些调用 `next` 方法的方法被称为 **消费适配器**（*consuming adaptors*），因为调用它们会消耗迭代器。一个消费适配器的例子是 `sum` 方法。这个方法获取迭代器的所有权并反复调用 `next` 来遍历迭代器，因而会消费迭代器。当其遍历每一个项时，它将每一个项加总到一个总和并在迭代完成时返回总和。

```
#[test]
fn iterator_sum() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    let total: i32 = v1_iter.sum();

    assert_eq!(total, 6);
}
```

迭代器适配器

`Iterator` trait 中定义了另一类方法，被称为 **迭代器适配器**（*iterator adaptors*），它们允许我们将当前迭代器变为不同类型的迭代器。可以链式调用多个迭代器适配器。不过因为所有的迭代器都是惰性的，必须调用一个消费适配器方法以便获取迭代器适配器调用的结果。

示例 13-14 展示了一个调用迭代器适配器方法 `map` 的例子，该 `map` 方法使用闭包来调用每个元素以生成新的迭代器。这里的闭包创建了一个新的迭代器，对其中 vector 中的每个元素都被加 1。：

```
let v1: Vec<i32> = vec![1, 2, 3];

let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();

assert_eq!(v2, vec![2, 3, 4]);
```

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 中最常见的指针是第四章介绍的 **引用**（*reference*）。引用以 `&` 符号为标志并借用了它们所指向的值。除了引用数据没有任何其他特殊功能，也没有额外开销。

另一方面，**智能指针**（*smart pointers*）是一类数据结构，它们的表现类似指针，但是也拥有额外的元数据和功能。

智能指针通常使用结构体实现。智能指针不同于结构体的地方在于其实现了 `Deref` 和 `Drop` trait。`Deref` trait 允许智能指针结构体实例表现的像引用一样，这样就可以编写既用于引用、又用于智能指针的代码。`Drop` trait 允许我们自定义当智能指针离开作用域时运行的代码。

Box< T >

最简单直接的智能指针是 *box*，其类型是 `Box<T>`。box 允许你将一个值放在堆上而不是栈上。留在栈上的则是指向堆数据的指针。

除了数据被储存在堆上而不是栈上之外，box 没有性能损失。不过也没有很多额外的功能。它们多用于如下场景：

- 当有一个在编译时未知大小的类型，而又想要在需要确切大小的上下文中使用这个类型值的时候
- 当有大量数据并希望在确保数据不被拷贝的情况下转移所有权的时候
- 当希望拥有一个值并只关心它的类型是否实现了特定 trait 而不是其具体类型的时候

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

递归类型

递归类型（*recursive type*）的值可以拥有另一个同类型的值作为其自身的一部分。但是这会产生一个问题，因为 Rust 需要在编译时知道类型占用多少空间。递归类型的值嵌套理论上可以无限地进行下去，所以 Rust 不知道递归类型需要多少空间。因为 box 有一个已知的大小，所以通过在循环类型定义中插入 box，就可以创建递归类型了。

cons list

```
(1, (2, (3, Nil)))
```

cons list 的每一项都包含两个元素：当前项的值和下一项。其最后一项值包含一个叫做 `Nil` 的值且没有下一项。cons list 通过递归调用 `cons` 函数产生。代表递归的终止条件（base case）的规范名称是 `Nil`，它宣布列表的终止。

```
enum List {
    Cons(i32, List),
    Nil,
}
fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

计算非递归类型的大小

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

当 Rust 需要知道要为 `Message` 值分配多少空间时，它可以检查每一个成员并发现 `Message::Quit` 并不需要任何空间，`Message::Move` 需要足够储存两个 `i32` 值的空间，依此类推。因为 enum 实际上只会使用其中的一个成员，所以 `Message` 值所需的空间等于储存其最大成员的空间大小。

使用 Box< T > 给递归类型一个已知的大小

因为 `Box<T>` 是一个指针，我们总是知道它需要多少空间：指针的大小并不会根据其指向的数据量而改变。

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Deref trait

实现 `Deref` trait 允许我们重载 **解引用运算符**（*dereference operator*） `*` （不要与乘法运算符或通配符相混淆）。通过这种方式实现 `Deref` trait 的智能指针可以被当作常规引用来对待，可以编写操作引用的代码并用于智能指针。

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

像引用一样使用 Box< T >

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

自定义智能指针


```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

通过实现 Deref trait 将某类型像引用一样处理


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

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

函数和方法的隐式 Deref 强制转换

Deref 强制转换（*deref coercions*）将实现了 `Deref` trait 的类型的引用转换为另一种类型的引用。例如，Deref 强制转换可以将 `&String` 转换为 `&str`，因为 `String` 实现了 `Deref` trait 因此可以返回 `&str`。Deref 强制转换是 Rust 在函数或方法传参上的一种便利操作，并且只能作用于实现了 `Deref` trait 的类型。当这种特定类型的引用作为实参传递给和形参类型不同的函数或方法时将自动进行。这时会有一系列的 `deref` 方法被调用，把我们提供的类型转换成了参数所需的类型。

Deref 强制转换的加入使得 Rust 程序员编写函数和方法调用时无需增加过多显式使用 `&` 和 `*` 的引用和解引用。这个功能也使得我们可以编写更多同时作用于引用或智能指针的代码。

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}
---
fn main() {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}
```

Drop trait

对于智能指针模式来说第二个重要的 trait 是 `Drop`，其允许我们在值要离开作用域时执行一些代码。可以为任何类型提供 `Drop` trait 的实现，同时所指定的代码被用于释放类似于文件或网络连接的资源。

指定在值离开作用域时应该执行的代码的方式是实现 `Drop` trait。`Drop` trait 要求实现一个叫做 `drop` 的方法，它获取一个 `self` 的可变引用。

```
struct CustomSmartPointer {
    data: String,
}
impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

通过 std::mem::drop 提早丢弃值


```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    c.drop();
    println!("CustomSmartPointer dropped before the end of main.");
}
```

这段代码不可运行

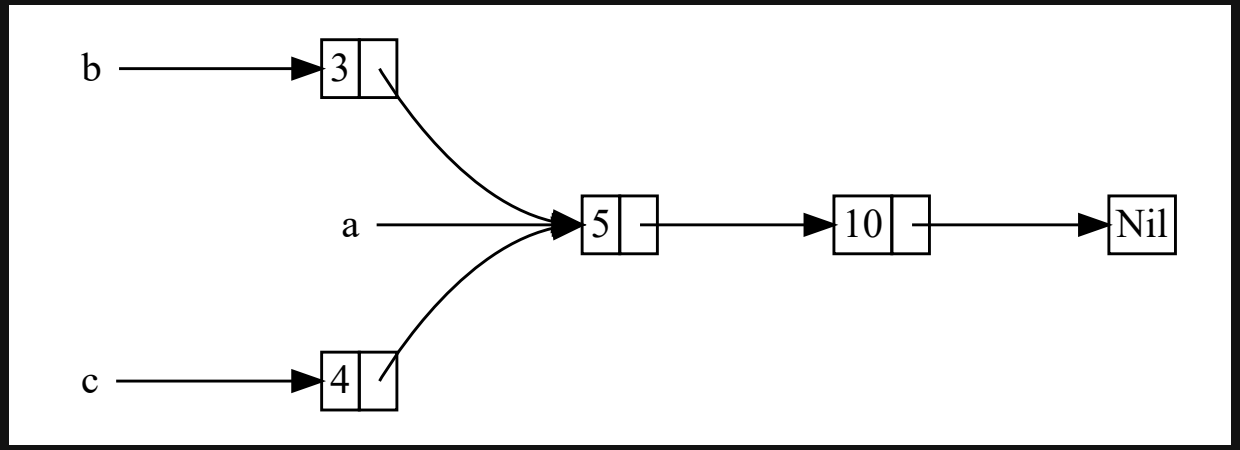
```
fn main() {
    let c = CustomSmartPointer {
        data: String::from("some data"),
    };
    println!("CustomSmartPointer created.");
    drop(c);
    println!("CustomSmartPointer dropped before the end of main.");
}
```

Rc < T >

大部分情况下所有权是非常明确的：可以准确地知道哪个变量拥有某个值。然而，有些情况单个值可能会有多个所有者。例如，在图数据结构中，多个边可能指向相同的节点，而这个节点从概念上讲为所有指向它的边所拥有。节点在没有任何边指向它从而没有任何所有者之前，都不应该被清理掉。

为了启用多所有权需要显式地使用 Rust 类型 `Rc<T>`，其为 **引用计数**（*reference counting*）的缩写。引用计数意味着记录一个值的引用数量来知晓这个值是否仍在被使用。如果某个值有零个引用，就代表没有任何有效引用并可以被清理。

使用 Rc< T > 共享数据



列表 `a` 包含 5 之后是 10，之后是另两个列表：`b` 从 3 开始而 `c` 从 4 开始。`b` 和 `c` 会接上包含 5 和 10 的列表 `a`。换句话说，这两个列表会尝试共享第一个列表所包含的 5 和 10。

使用 Rc< T > 共享数据

尝试使用 `Box<T>` 定义的 `List` 实现并不能工作

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

`Cons` 成员拥有其储存的数据，所以当创建 `b` 列表时， `a` 被移动进了 `b` 这样 `b` 就拥有了 `a` 。接着当再次尝试使用 `a` 创建 `c` 时，这不被允许，因为 `a` 的所有权已经被移动。

使用 Rc< T > 共享数据


```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::rc::Rc;

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

克隆 Rc< T > 会增加引用计数

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

在程序中每个引用计数变化的点，会打印出引用计数，其值可以通过调用 `Rc::strong_count` 函数获得。

通过不可变引用， `Rc<T>` 允许在程序的多个部分之间只读地共享数据。如果 `Rc<T>` 也允许多个可变引用，则会违反第四章讨论的借用规则之一：相同位置的多个可变借用可能造成数据竞争和不一致。

RefCell< T > 和内部可变性模式

内部可变性（*Interior mutability*）是 Rust 中的一个设计模式，它允许你即使在有不可变引用时也可以改变数据，这通常是借用规则所不允许的。为了改变数据，该模式在数据结构中使用 `unsafe` 代码来模糊 Rust 通常的可变性和借用规则。不安全代码表明我们在手动检查这些规则而不是让编译器替我们检查。

不同于 `Rc<T>`，`RefCell<T>` 代表其数据的唯一的所有权。那么是什么让 `RefCell<T>` 不同于像 `Box<T>` 这样的类型呢？回忆一下第四章所学的借用规则：

1. 在任意给定时刻，只能拥有一个可变引用或任意数量的不可变引用 **之一**（而不是两者）。
2. 引用必须总是有效的。

对于引用和 `Box<T>`，借用规则的不可变性作用于编译时。对于 `RefCell<T>`，这些不可变性作用于 **运行时**。对于引用，如果违反这些规则，会得到一个编译错误。而对于 `RefCell<T>`，如果违反这些规则程序会 panic 并退出。

内部可变性：不可变值的可变借用

借用规则的一个推论是当有一个不可变值时，不能可变地借用它。例如，如下代码不能编译：

```
fn main() {
    let x = 5;
    let y = &mut x;
}
```

然而，特定情况下，令一个值在其方法内部能够修改自身，而在其他代码中仍视为不可变，是很有用的。值方法外部的代码就不能修改其值了。`RefCell<T>` 是一个获得内部可变性的方法。`RefCell<T>` 并没有完全绕开借用规则，编译器中的借用检查器允许内部可变性并相应地在运行时检查借用规则。如果违反了这些规则，会出现 panic 而不是编译错误。

RefCell< T > 在运行时记录借用

当创建不可变和可变引用时，我们分别使用 `&` 和 `&mut` 语法。对于 `RefCell<T>` 来说，则是 `borrow` 和 `borrow_mut` 方法，这属于 `RefCell<T>` 安全 API 的一部分。`borrow` 方法返回 `Ref<T>` 类型的智能指针，`borrow_mut` 方法返回 `RefMut<T>` 类型的智能指针。这两个类型都实现了 `Deref`，所以可以当作常规引用对待。

```
    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            let mut one_borrow = self.sent_messages.borrow_mut();
            let mut two_borrow = self.sent_messages.borrow_mut();

            one_borrow.push(String::from(message));
            two_borrow.push(String::from(message));
        }
    }
```

结合 Rc< T > 和 RefCell< T > 来拥有多个可变数据所有者

```rust
#[derive(Debug)]
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main() {
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10;

    println!("a after = {:?}", a);
    println!("b after = {:?}", b);
    println!("c after = {:?}", c);
```

并发编程

并发编程（*Concurrent programming*），代表程序的不同部分相互独立地执行

并行编程（*parallel programming*）代表程序不同部分同时执行

并行包含于并发，并发包括并行

对于操作系统：

- 进程是资源分配的基本单位
- 线程是任务执行的基本单位

并发编程（多线程）的目的：**将一批任务分配给多个线程同时处理**

使用 spawn 创建新线程


```rust
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));
    }
}
```

使用 join 等待所有线程结束


```rust
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();
}
```

将 move 闭包与线程一同使用

```
use std::thread;

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

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

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

```
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();
}
```

使用消息传递在线程间传送数据

为了实现消息传递并发，Rust 标准库提供了一个 **信道**（*channel*）实现。信道是一个通用编程概念，表示数据从一个线程发送到另一个线程。

编程中的信息渠道（信道）有两部分组成，一个发送者（transmitter）和一个接收者（receiver）。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信道被 **关闭**（*closed*）了。

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

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);
}
```

mpsc：**多个生产者，单个消费者**（*multiple producer, single consumer*）

信道与所有权转移

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

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

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

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

`send` 函数获取其参数的所有权并移动这个值归接收者所有。这可以防止在发送后再次意外地使用这个值；所有权系统检查一切是否合乎规则。

发送多个值并观察接收者的等待


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

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

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

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

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

通过克隆发送者来创建多个生产者


```
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));
    }
```

共享状态并发

在某种程度上，任何编程语言中的信道都类似于单所有权，因为一旦将一个值传送到信道中，将无法再使用这个值。共享内存类似于多所有权：多个线程可以同时访问相同的内存位置。

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。

互斥器以难以使用著称，因为你不得不记住：

1. 在使用数据之前尝试获取锁。
2. 处理完被互斥器所保护的数据之后，必须解锁数据，这样其他线程才能够获取锁。

Mutex< T >


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

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

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

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

在线程间共享 Mutex< T >


```rust
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());
}
```

多线程和多所有权


```rust
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());
}
```

原子引用计数 Arc< T >

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

对应 rustlings 练习

余下