程序设计训练之 Rust 编程语言

第六讲：并发编程

韩文弢

清华大学计算机科学与技术系

2022 年 9 月 5 日

1

闭包

闭包的概念

- 函数式语言经常会用到闭包 (closure)、匿名函数或者是 lambda 函数的编程范式。
- 在 Rust 语言中，闭包与所有权模型是匹配的。

```
let square = |x: i32| -> i32 { x * x };
println!("{}", square(3));
// => 9
```

- 内联的函数定义可以绑定到变量上。当闭包被调用时，将会执行函数体。

闭包的语法

```rust
let foo_v1 = |x: i32| { x * x };
let foo_v2 = |x: i32, y: i32| x * y;
let foo_v3 = |x: i32| {
    // Very Important Arithmetic
    let y = x * 2;
    let z = 4 + y;
    x + y + z
};
let foo_v4 = |x: i32| if x == 0 { 0 } else { 1 };
```

- 闭包定义的语法与函数定义相近。
- 在 `||` 之间指定参数，之后是返回表达式。
  - 返回表达式可以是 `{}` 里的一组表达式。

- 使用 `let` 代替 `fn`
- 参数在两条竖线之间
- 花括号是可选的

类型推导

```rust
let square_v4 = |x: u32| { (x * x) as i32 };

let square_v4 = |x| -> i32 { x * x }; // unable to infer enough
let square_v4 = |x|        { x * x }; // type information!
```

- 与函数不同，闭包**不需要**指定参数或返回值的类型。
  - 但是在编译器无法推导出正确的类型时，仍然可以显式指定。
- 对于函数来说，指定类型可以使得函数的定义更加清楚。而对闭包而言，易用性更加重要。

闭包的环境

- 闭包可以包含它所在的环境。

```
let magic_num = 5;
let magic_johnson = 32;
let plus_magic = |x: i32| x + magic_num;
```

- 即使没有传参，闭包 `plus_magic` 也可以引用 `magic_num`。
  - `magic_num` 在该闭包的环境中。
  - `magic_johnson` 没有被借用。

闭包与借用

- 如果在闭包绑定后试图以冲突的方式借用，会发生编译错误。

```
let mut magic_num = 5;
let magic_johnson = 32;
let plus_magic = |x: i32| x + magic_num;

let more_magic = &mut magic_num; // Err!
println!("{}", magic_johnson); // Ok!
```

闭包与借用的错误

```
error: cannot borrow `magic_num` as mutable because it is
already borrowed as immutable

[...] the immutable borrow prevents subsequent moves or mutable
borrows of `magic_num` until the borrow ends
```

- `plus_magic` 在创建闭包的时候借用了 `magic_num`。
- `magic_johnson` 没有在闭包里用到，因此它的所有权不受影响。

闭包的环境

- 此类错误可以通过使闭包超过作用域来修复。

```
let mut magic_num = 5;
{
    let plus_magic = |x: i32| x + magic_num;
} // the borrow of magic_num ends here

let more_magic = &mut magic_num; // Ok!
println!("magic_num: {}", more_magic);
```

移动闭包

- 闭包也可以选择所有权的处理方式。
- 有时候借用不能满足需求。
- 可以使用 `move` 关键字强制闭包获得环境变量的所有权。
  - 对于 `Copy` 类型来说，“获得所有权”表示拷贝，不一定真的是转移所有权。

```
let mut magic_num = 5;
let own_the_magic = move |x: i32| x + magic_num;
let more_magic = &mut magic_num;
```

移动闭包的应用场景

- 在闭包 `f` 需要比创建它的作用域活得更久时，需要用到移动闭包。
  - 例如，将 `f` 传递给一个线程，或者从一个函数返回 `f`。
  - `move` 会禁止将引用带入闭包。

```
fn make_closure(x: i32) -> Box<Fn(i32) -> i32> {
    let f = move |y| x + y;
    Box::new(f)
}

let f = make_closure(2);
println!("{}", f(3));
```

闭包与所有权

- 有时候，闭包**必须**获得环境变量的所有权以确保合法性。在以下情况，这种行为会自动发生（不使用 `move`）：
  - 如果有值被移动到返回值里。

```
let lottery_numbers = vec![11, 39, 51, 57, 75];
{
    let ticket = || { lottery_numbers };
}
// The braces do no good here.
println!("{:?}", lottery_numbers); // use of moved value
```

  - 或者移动到其他别的地方。

```
let numbers = vec![2, 5, 32768];
let alphabet_soup = || { numbers; vec!['a', 'b'] };
                        // ^ throw away unneeded ingredients
println!("{:?}", numbers); // use of moved value
```

- 如果类型不是 `Copy` 的，原来的变量会失效。

闭包与所有权：只能调用一次的情况

```
let numbers = vec![2, 5, 32768];
let alphabet_soup = || { numbers; vec!['a', 'b'] };
                        // ^ throw away unneeded ingredients
alphabet_soup();
alphabet_soup(); // use of moved value
```

- 如果闭包拥有数据，并且在执行过程中发生了所有权的转移，那么这样的闭包只能被调用一次。
  - 这种情况是隐含 **move** 行为的，因为只有拥有 **numbers** 的所有权才能转移。

闭包与所有权：可以多次调用的情况

```
let numbers = vec![2, 5, 32768];
let alphabet_soup = move || { println!("{:?}", numbers) };
alphabet_soup();
alphabet_soup(); // Delicious soup
```

- 而拥有数据所有权但是不转移的闭包是可以多次调用的。
- 这种闭包也可以通过引用来实现。

闭包特型

- 闭包根据所有权情况不同分别实现下列三种特型：Fn、FnMut、FnOnce
  - 实现这些特型的类型可以进行方法调用操作（重载方法调用操作符 ()）。

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call"
      fn call_mut(&mut self, args: Args) -> Self::Output;
}
pub trait FnOnce<Args> {
    type Output;
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

闭包特型的解释

- 这三种特型的区别在于处理 `self` 的方式：
  - `Fn` 借用 `self`，也就是使用 `&self`。
  - `FnMut` 以可变方式借用 `self`，也就是使用 `&mut self`。
  - `FnOnce` 获得 `self` 的所有权。
- 因此，`Fn` 是 `FnMut` 的超集，`FnMut` 是 `FnOnce` 的超集。
- 函数也实现了这类特型。
- 实际上 `|| {}` 表示闭包的语法是一种语法糖。Rust 会为闭包的环境生成一个结构体，实现相应的特型，然后使用。

将闭包作为参数

- 可以像函数指针一样传递闭包。
- 例如以下简化版本的 `map` 函数：

```
// self = Vec<A>
fn map<A, B, F>(self, f: F) -> Vec<B>
    where F: FnMut(A) -> B;
```

- `map` 有一个 `f: F` 参数，这里 `F` 是实现 `FnMut` 特型的类型。
- 可以使用普通函数，因为普通函数实现了 `FnMut` 特型。

返回闭包

- 有时候需要将闭包作为函数的返回值。
- 但是闭包是特型对象，它的大小是不确定的。

```
fn i_need_some_closure() -> (Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

```
error: the trait `core::marker::Sized` is not implemented
    for the type `core::ops::Fn(i32) -> i32 + 'static`
```

返回闭包

- 因此需要通过间接方式来返回闭包。

```
fn i_need_some_closure_by_reference() -> &dyn (Fn(i32) -> i32) {
    let local = 2;
    |x| x * local
}
```

```
error: missing lifetime specifier
```

- 没有给闭包指定生命周期。原因是这里返回的引用会比闭包本身活得更久，变成了悬垂指针。

返回闭包

- 使用 Box 将其变为动态的生命周期。

```
fn box_me_up_that_closure() -> Box<dyn Fn(i32) -> i32> {
    let local = 2;
    Box::new(|x| x * local)
}
```

```
error: closure may outlive the current function, but it
borrows `local`, which is owned by the current function [E0373]
```

- 要返回的闭包依赖它的环境，当它被返回后，local 已经销毁。

返回闭包

- 通过移动语法来修正这个问题。

```
fn box_up_your_closure_and_move_out() -> Box<dyn Fn(i32) -> i32> {
    let local = 2;
    Box::new(move |x| x * local)
}
```

2

并发的概念

什么是并发?

- 并发 (concurrency) 是指程序同时有多个正在运行的线程 (threads)。
- 线程之间可以共享数据，而不引入通信（如网络、进程间通信等）的开销。
- 线程比单独的进程要轻量级。
  - 在线程之间切换的时候不会发生大开销的操作系统上下文切换动作。
- 注意并发和并行 (parallelism) 之间的区别。

什么是线程?

- 线程是指令执行的上下文，以及对一些数据的引用关系（可能是共享的）。
- 上下文包括一组寄存器的值、一个栈，以及其他与当前执行上下文相关的信息。
- 每个程序至少有一个线程。
- 有一个线程调度器 (scheduler) 来管理线程的执行，决定什么时候运行哪个线程。
- 程序可以创建新的线程，由调度器负责运行。

并发执行

- 考虑下面的类 Rust 代码（忽略所有权等方面的问题）:

```
let mut x = 0;

fn foo() {
    let mut y = &mut x;
    *y = 1;
    println!("{}", *y); // foo expects 1
}

fn bar() {
    let mut z = &mut x;
    *z = 2;
    println!("{}", *z); // bar expects 2
}
```

指令的交错执行

- 假设有两个线程，一个执行 `foo`，另一个执行 `bar`。
- 调度器可以以任意方式交错地执行指令。
- 因此，一种可能的执行顺序是：

```
/* foo */ let mut y = &mut x;
/* foo */ *y = 1;
/* foo */ println!("{}", *y); // foo expects 1
          // => 1
/* bar */ let mut z = &mut x;
/* bar */ *z = 2;
/* bar */ println!("{}", *z); // bar expects 2
          // => 2
```

- 如果是这样执行，结果是预料中的。

指令的交错执行

- 然而，这样的执行顺序不是每次都能保证的，甚至都不一定会发生。
- `foo` 和 `bar` 可能会以任意的方式交错执行，产生预料之外的结果：

```
/* bar */ let mut z = &mut x;
/* bar */ *z = 2;
/* foo */ let mut y = &mut x;
/* foo */ *y = 1;
/* bar */ println!("{}", *z); // bar expects 2
          // => 1
/* foo */ println!("{}", *y); // foo expects 1
          // => 1
```

- 需要一定的机制来确保关键事件能够按一定的顺序发生，这样可以产生预料之中的结果。

并发编程的难点

- **数据共享**：如果两个线程同时试图改写同一份数据，会产生什么结果?
  - 例如之前那个例子中对 `x` 的写入操作。
- **数据竞争**：同一段代码的行为与它的执行情况有关。
  - 例如之前那个例子中对 `x` 的读取操作。

并发编程的难点（续）

- **同步**：如何保证所有线程都有正确的“世界观”？
  - 例如，有一组线程共享同一个缓冲区，每个线程 `i` 会去写 `buffer[i]`，然后试图读取整个缓冲区来决定下一步动作。
  - 在线程之间发送数据时，如何确认其他线程在正确的地方收到了数据。
- **死锁**：如果在线程间安全地共享资源，确保线程不会相互锁定数据访问行为？

死锁

- 当多个线程访问某一共享资源时，可能会发生死锁。死锁发生后，所有参与的线程都无法访问数据。
- 死锁发生有四个条件：
  - 互斥：资源以非共享的模式锁定。
  - 持有资源：线程持有一个资源，并去要求获得其他线程持有的资源。
  - 非抢占：持有资源的线程不会自愿释放资源。
  - 等待成环：等待资源的线程之间形成环状关系。
- 为了避免死锁发生，只要打破上述条件之一。

哲学家就餐问题

- 一个经典的死锁问题。
- N 个哲学家坐在一张圆桌周围，交替地进行吃饭和思考。
- 每个哲学家需要一双筷子用来吃饭，但是一共只有 N 根筷子，每两个哲学家之间有一根。
- 哲学家的行为用算法描述如下：
  - 拿起他左侧的那根筷子（获取一个资源的锁）。
  - 拿起他右侧的那根筷子（获取一个资源的锁）。
  - 吃饭（使用资源）。
  - 将两根筷子放回原处（释放资源的锁）。

哲学家就餐问题（续）

- 按照上述行为，会发生什么情况?
- 以 $N = 3$ 为例：
  - 1 号哲学家拿起他左侧的那根筷子。
  - 2 号哲学家拿起他左侧的那根筷子。
  - 3 号哲学家拿起他左侧的那根筷子。
  - 三位哲学家都试图去拿他们右侧的那根筷子，然后卡在这一步骤，原因是筷子都在手里了。
- 如何改进来避免死锁?

3

线程

Rust 的线程

- Rust 的标准库提供了线程 `std::thread`。
- 每个线程有自己的栈和状态。
- 使用闭包来指定线程的行为：

```rust
use std::thread;

thread::spawn(|| {
    println!("Hello, world!");
});
```

线程句柄

- thread::spawn 返回 JoinHandler 类型的线程句柄 (handlers)。

```rust
use std::thread;

let handle = thread::spawn(|| {
    "Hello, world!"
});

println!("{:?}", handle.join());
// => Ok("Hello, world!")
```

- join() 会阻塞当前线程的执行，直到句柄对应的线程终止。
- join() 返回对应线程返回值的 Ok，或者是恐慌值的 Err。

std::thread::JoinHandler

- 当线程的句柄被丢弃时，线程会变为失联 (detached) 状态。此时，它还在运行。
- 不能克隆线程句柄，只有一个变量有权限来做线程的汇合 (join)。

panic!

- 如果线程发生恐慌，那么将无法从该线程内部来进行恢复。
- Rust 的线程如果发生恐慌，和创建这个线程的线程没有关系。
  - 只有发生恐慌的线程会崩溃。
  - 这个线程会进行相关的清理工作，包括栈和其他资源。
  - 其他线程能够读取恐慌的消息。
- 如果主线程恐慌或者正常结束，其他线程也会被终止。
  - 主线程可以在自己结束之前去等待其他线程结束。

std::thread::Thread

- 当前正在运行的线程可以调用 thread::park() 来暂停自己的执行。
- 之后可以通过线程的 unpark() 方法来继续执行。

```
use std::thread;

let handle = thread::spawn(|| {
    thread::park();
    println!("Good morning!");
});
println!("Good night!");
handle.thread().unpark();
```

- JoinHandler 提供了 thread() 方法来获取对应线程的 Thread 对象。
- 当前正在运行的线程可以通过 thread::current() 获得。

多个线程

- 可以创建多个线程：

```rust
use std::thread;

for i in 0..10 {
    thread::spawn(|| {
        println!("I'm first!");
    });
}
```

线程与所有权

- 创建线程时同样需要遵守所有权规则（包括闭包与所有权的规则）:

```rust
use std::thread;

for i in 0..10 {
    thread::spawn(|| {
        println!("I'm #{}!", i);
    });
}
// Error!
// closure may outlive the current function, but it borrows `i`,
// which is owned by the current function
```

线程与所有权

- 这里，线程所执行的闭包要拥有 i。
- 因此用 move 来创建移动闭包，获得所有权。

```rust
use std::thread;

for i in 0..10 {
    thread::spawn(move || {
        println!("I'm #{}!", i);
    });
}
```

4

共享线程状态

Send 与 Sync

- Rust 的类型系统包含要求满足并发承诺的特型：
  - `Send`：表示可以在线程间安全转移的类型。
  - `Sync`：表示可以在线程间（通过引用）安全共享的类型。
- 以上两种特型都是标记特型，本身不提供方法。

Send

```
pub unsafe trait Send { }
```

- Send 类型可以将它的所有权在线程间转移。
- 如果一种类型没有实现 Send，那么它只能留在原来的线程里。
  - 例如，Rc 没有实现 Send。

Sync

```
pub unsafe trait Sync { }
```

- Sync 类型（通过引用）在多个线程间使用时不会引发内存安全问题。
- 所有基本类型是 Sync 的，所有只含有 Sync 类型的复合类型是 Sync 的。
  - 不可变类型 (&T) 和简单的继承可变类型 (Box<T>) 是 Sync 的。
  - 所有不带内部可变性的类型会自动获得 Sync。

Sync

- 类型 T 是 Sync 的，当且仅当类型 &T 是 Send 的。
  - 也就是说，如果引用 &T 可以在线程间传递，那么 T 就可以在线程间共享。
- 作为一个推论，如果 T 是 Sync 的，那么 &mut T 也是 Sync 的。
- 具有内部可变性的类型不是 Sync 的。
  - 例如，Cell<T> 在不可变引用时仍然能够修改其中的值。

不安全性

- 尽管没有需要实现的方法，`Send` 和 `Sync` 的实现都是 `unsafe` 的。
- 将一种特型标注为 `unsafe` 表示实现这种特型需要由程序员来保证特型的承诺。
  - 也就是说，编译器无法检查特型的承诺是否满足。
- `Send` 和 `Sync` 是不安全的，其原因是线程安全性无法由 Rust 编译器来保证。
  - 事实上，100% 的线程安全性只能靠不使用线程在保证。
- 因此，`Send` 和 `Sync` 需要由安全代码不能提供的一层信任来给出。

共享线程状态示例

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

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

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}

// error: capture of moved value: `data`
//         data[i] += 1;
//         ^~~~
```

共享线程状态示例中的问题

- 如果每个线程都要 `data` 所有权，会导致 `data` 有多个所有者。
- 为了共享 `data`，需要能够在线程间安全共享的类型。
  - 也就是需要 `Sync` 的类型。

std::sync::Arc<T>

- 解决方案：使用 `Arc<T>`，原子性的引用计数指针 (**A**tomic **R**eference-**C**ounted pointer)。
  - 和 `Rc` 很像，但是通过原子性的计数来保证线程安全性。
  - 也有一种对应的 `Weak` 类型。

共享线程状态示例

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

fn main() {
    let mut data = Arc::new(vec![1, 2, 3]);

    for i in 0..3 {
        let data = data.clone(); // Increment `data`'s ref count
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

共享线程状态示例中的问题

- 修改后还是没法通过编译。

```
error: cannot borrow immutable borrowed content as mutable
                    data[i] += 1;
                    ^~~~
```

- 与 `Rc` 一样，`Arc` 也不具有内部可变性。
- 它的内容只有当仅存在一个强引用且没有弱引用时才能修改。
  - 而克隆 `Arc` 会违反这个要求。

共享线程状态示例中的问题（续）

- `Arc<T>` 假设它的内容是 `Sync` 的，因此无法修改。
- 此时也不能使用 `RefCell`，因为 `RefCell` 不是线程安全的。
- 需要使用 `Mutex<T>` 来解决这个问题。

互斥锁

- 互斥锁 (mutexes) 是 **Mut**ual **Ex**clusion 的缩写。
- 互斥锁保证它所包含的值在同一时刻只有一个线程能够访问。
- 为了访问由互斥锁保护的数据，需要获得互斥锁中的锁。
- 如果有人正拿着锁，那么其他人可以选择放弃并在后面再尝试，或者阻塞等待直到锁被释放。

std::sync::Mutex<T>

- 用 Mutex 包裹一个值时，需要调用 lock 方法来获取对值的访问权限。
  - lock 方法返回一个 LockResult。
- 如果互斥锁是锁定的状态，lock 方法会阻塞等待，直到锁被释放。
  - 也可以使用 try_lock 方法避免阻塞等待。锁定成功后可以得到一个 MutexGuard，通过解引用来访问里面的 T 类型数据。

互斥锁的中毒状态

- 如果一个线程锁定了一个互斥锁，然后发生了恐慌，此时该互斥锁会进入**中毒** (poisoned) 状态，因为这个锁不会被释放了。
- 因此 `lock()` 返回的是一个 `LockResult`：
  - 如果是 `Ok(MutexGuard)`，那么互斥锁没有中毒，可以正常使用。
  - 如果是 `Err(PoisonError<MutexGuard>)`，那么互斥锁处于中毒状态。

共享线程状态示例

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

fn main() {
    let mut data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone(); // Increment `data`'s ref count
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep(Duration::from_millis(50));
}
```

5

通道

通道

- 通道 (channels) 可以用来同步线程之间的状态。
- 通道能够在线程之间传递消息。
- 可以用来提醒其他线程关于数据已经就绪、事件已经发生等情况。

std::sync::mpsc

- 实现多生产者、单消费者的通信功能 (**M**ulti-**P**roducer, **S**ingle-**C**onsumer)。
- 涉及三种主要类型：
  - `Sender`
  - `SyncSender`
  - `Receiver`
- 其中，`Sender` 或 `SyncSender` 用于给 `Receiver` 发送数据。
- `Sender` 类型可以克隆，交给多个线程实现**多生产者**。
- `Receiver` 类型不能克隆，因此是**单消费者**。

std::sync::mpsc

- 使用 `channel<T>()` 函数来创建一对连接的 `(Sender<T>, Receiver<T>)`。
- `Sender` 是**异步**通道，发送数据时不会阻塞发送线程。
  - 相当于 `Sender` 有一个无限大的缓冲区。
- 试图在接收线程从 `Receiver` 接收数据时，如果数据还没到，会发生阻塞。

std::sync::mpsc

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

fn main() {
    let (tx, rx) = channel();
    for i in 0..10 {
        let tx = tx.clone();
        thread::spawn(move|| {
            tx.send(i).unwrap();
        });
    }
    drop(tx);

    let mut acc = 0;
    while let Ok(i) = rx.recv() {
        acc += i;
    }
    assert_eq!(acc, 45);
}
```

std::sync::mpsc

- 使用 sync_channel<T>() 函数来创建一对连接的 (SyncSender<T>, Receiver<T>)。
- SyncSender 是**同步**的，发送消息时会发生阻塞。
  - 相当于 SyncSender 有一个有限的缓冲区，如果缓冲区满了发送就会阻塞。
- Receiver 和之前是一样的，因此也会阻塞。

std::sync::mpsc

- 通道的发送/接收操作都会返回一个 `Result`。
- 如果通道的另一端挂起，操作会发生错误。
- 一旦通道断开，就无法重新连接。

并发编程的难点

- **数据共享**：通过 Send、Sync 来标记共享的行为，使用 Arc 和 Mutex 来实现共享。
- **数据竞争**：Sync
- **同步**：可以通过通道来实现通信。
- **死锁**：有什么办法可以解决?

哲学家就餐问题的改进

- 就目前所提供的并发基本操作，无法保证没有死锁的情况。
- 也就是说，尽管可以确保安全性，但还是无法避免所有的逻辑问题（例如死锁）。
- 没有统一的解决办法。
- 对于哲学家就餐问题，有一些具体的办法：
  - 最后一个哲学家用相反的方向来拿筷子。
  - 在哲学家之间传递一个令牌，拿到令牌的哲学家才可以去尝试拿筷子。

6

小结

本讲小结

- 闭包
- 并发的概念
- 线程
- 共享线程状态
- 通道