程序设计训练（Rust 语言）

第 6 讲：并发编程

韩文弢

清华大学

2024—2025 学年度夏季学期

本讲内容

6.1 闭包

6.1.1 闭包的引入

很多时候在使用某个功能时需要额外指定一些行为作为参数，例如排序时想使用特定的顺序。

```rust
struct City {
    name: String,
    population: i64,
    country: String,
    // ...
}
cities.sort();
```

按城市名字排序？人口排序？升序？降序？

6.1.2 闭包的引入（续）

可以用函数来指定行为，称为用户自定义函数（user-defined function, UDF）。

Rust

```rust
use std::cmp::Ordering;
// struct City
fn cmp_population_desc(a: &City, b: &City) -> Ordering {
    a.population.cmp(&b.population).reverse()
}
cities.sort_by(cmp_population_desc);
```

编写单独函数的不足：逻辑分散、不方便提供额外信息。

6.1.3 闭包的概念

Rust 语言提供了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。

```rust
// struct City
cities.sort_by(
    |a, b| a.population.cmp(&b.population).reverse()
);
```

6.1.4 闭包的语法

```rust
let foo_v1 = |x: i32| -> i32 { x * x };
let foo_v2 = |x: i32, y: i32| x * y;
let foo_v3 = |x: i32| {
    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 || 之间指定参数列表，之后是返回类型（可选）和返回表达式。
- 返回表达式可以是 {} 里的一组表达式。

6.1.5 类型推导

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

let square_v4 = |x| -> i32 { x * x };  // Compile error
let square_v4 = |x|        { x * x };  // Compile error
```

- 与函数不同，闭包**可以不**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，仍然需要显式指定。
- 设计决策
  - 对函数来说，可能需要作为跨模块的接口，指定类型可以使得函数的定义更加清楚。
  - 对闭包来说，就在定义的位置使用，易用性更加重要。

6.1.6 闭包与环境：变量捕获

- 闭包可以使用它所在环境中的值，称为变量捕获（variable capture）。

```rust
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` 没有被捕获。
- 捕获同样遵循 Rust 的规则，分为三种：不可变借用、可变借用、转移。

6.1.7 变量捕获：不可变借用示例

Rust

```rust
let s = "Hello".to_string();
let f = |t| s.clone() + t;
println!("{}", f("Alice"));  // ==> HelloAlice
println!("{}", f("Bob"));    // ==> HelloBob
```

6.1.8 变量捕获：可变借用示例

Rust

```rust
let mut s = "Hello".to_string();
let mut f = |t| {
    s.push_str(t);
    s.clone()
};
println!("{}", f("Alice"));  // ==> HelloAlice
println!("{}", f("Bob"));    // ==> HelloAliceBob
```

6.1.9 变量捕获：转移示例

Rust

```rust
let s = "Hello".to_string();
let f = |t| s + t;
println!("{}", f("Alice"));  // ==> HelloAlice
// Error: println!("{}", f("Bob"));
```

6.1.10 闭包的作用域

当闭包本身的生命周期超过环境时，需要用 move 关键字指定转移捕获。

```rust
let f;
{
    let x = 5;
    f = move |y| x + y;  // 必须加上 move
    println!("{}", f(1));
}
println!("{}", f(2));
```

6.1.11 闭包特型

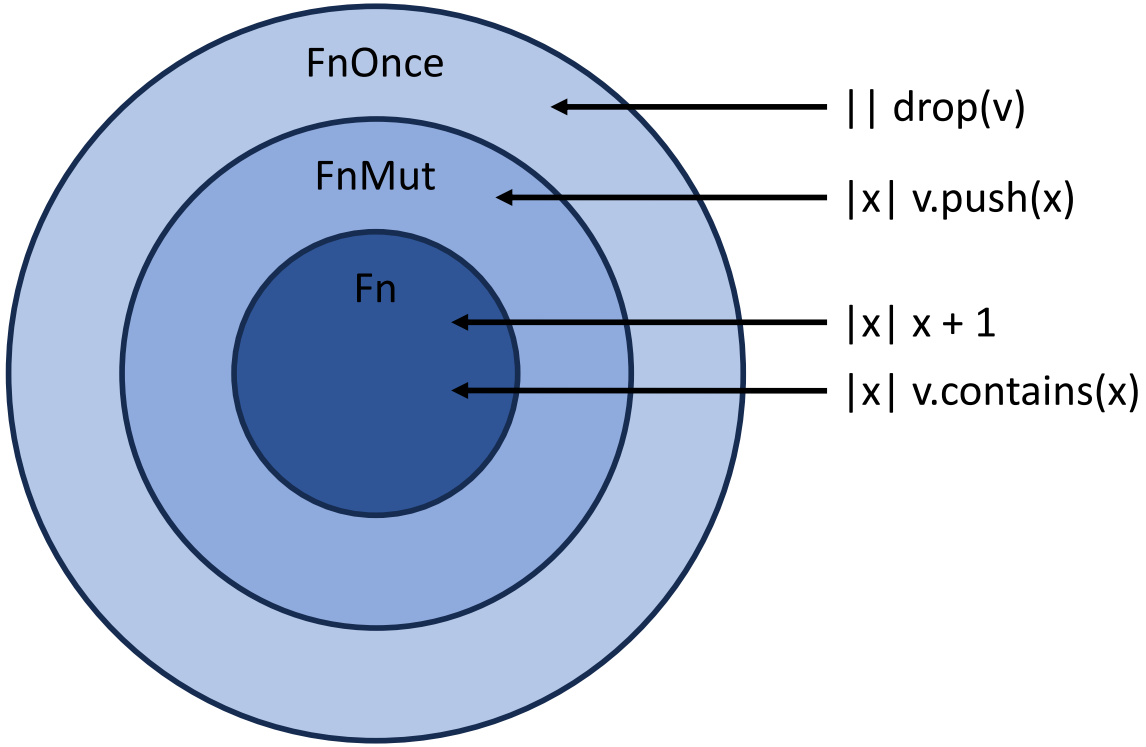
闭包根据所有权情况不同分别实现下列三种特型：Fn、FnMut、FnOnce

- 由编译器根据捕获方式自动确定。
- 实现这些特型的类型可以进行方法调用操作（重载操作符 ()）。

```rust
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;
}
```

6.1.12 闭包特型之间的关系

- `Fn`：不修改闭包的环境变量，可以反复调用。
- `FnMut`：修改闭包的环境变量，可以反复调用。
- `FnOnce`：获取闭包的环境变量的所有权，至少可以调用一次。



6.1.13 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` 的子集，`FnMut` 是 `FnOnce` 的子集。
- 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。

6.1.14 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：

```rust
// 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。

6.1.15 返回闭包 (1)

- 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。

```rust
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`
```

6.1.16 返回闭包 (2)

因此需要通过间接方式来返回闭包。

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

Rust

```
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。

6.1.17 返回闭包 (3)

使用 Box 将其变为动态的生命周期。

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

Rust

```
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。

6.1.18 返回闭包 (4)

通过强制转移语法（move 关键字）来修正这个问题。

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

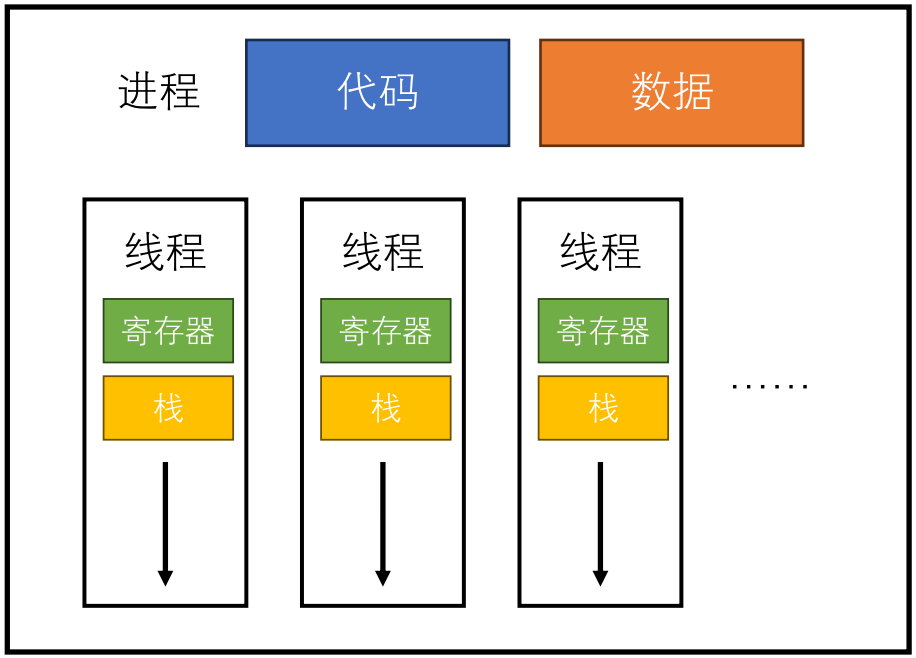
6.2 并发的概念

6.2.1 什么是并发?

- **并发** (concurrency) 是指一个程序同时有多个正在运行的指令序列，称为**线程** (threads)。
- 线程之间可以过共同的内存地址空间来共享数据，而不引入通信（如网络、进程间通信等）的开销。
- 线程比进程（processes）要轻量级。
  - 线程之间切换的代价比进程小，主要涉及操作系统上下文切换 (context switch)。
- 注意并发和并行 (parallelism) 之间的区别：并行要有多个执行单元同时运行多个指令序列，并发可以只有一个执行单元。

6.2.2 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、一个栈等。
- 每个程序（进程）至少有一个线程。
- 操作系统有线程调度器（scheduler）来管理线程的执行，决定什么时候运行哪个线程。



6.2.3 并发执行

考虑下面的类 Rust 代码（忽略所有权等方面的问题）：

Rust

```rust
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
}
```

6.2.4 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。一种可能的执行顺序是：

```rust
/* 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
```

6.2.5 指令的交错执行（续）

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：

```rust
/* 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
```

6.2.6 并发优点：数组求和

单线程计算

- 仅有单核参与计算，性能较低
- 编写简单，易于理解

```rust
fn foo(input: &i32) -> i32 {
    // Simulate a time-consuming operation
    thread::sleep(Duration::from_millis(1));
    input * 2
}

let arr: Vec<i32> = (1..=1000).collect();
let sum: i32 = arr.iter().map(foo).sum();
```

6.2.7 并发优点：数组求和（多线程）

多线程计算

- 多任务参与，在多核 CPU 上可以并行计算，提高性能
- 编写难度上升，还需考虑竞争与安全问题

Rust

```rust
let n_threads = 10;
let mut threads = Vec::new();
for i in 0..n_threads {
    let start = i * arr.len() / n_threads;
    let end = (i + 1) * arr.len() / n_threads;
    let thread = std::thread::spawn(move ||
        arr[start..end].iter().map(foo).sum::<i32>()
    );
    threads.push(thread);
}
let sum: i32 = threads.into_iter().map(|t| t.join().unwrap()).sum();
```

6.2.8 并发优点：数组求和（高级抽象）

现代多线程计算：高级抽象

- 使用 rayon 库，可以方便地进行并行计算
- 由库来管理线程和任务调度框架，屏蔽底层管理细节
- 降低编写难度，且保留了多线程的性能

Rust

```rust
// Import the rayon prelude to use parallel iterators
use rayon::prelude::*;

let arr: Vec<i32> = (1..=1000).collect();
// Use parallel iterators to sum the array in parallel
let sum: i32 = arr.par_iter().map(foo).sum();
```

6.2.9 并发编程技术

- 多线程编程
  - 使用 std::thread 库，可以手动操作、管理线程
  - 使用 rayon 库，可以在不关心线程管理的情况下进行并行计算
- SIMD (Single Instruction Multiple Data)：单指令多数据
  - 在一条指令中处理多个数据，如一条指令处理 4 个 32 位整数
  - 现代 CPU 的 SIMD 指令集可以显著提高计算性能
  - Rust 已经开始尝试加入 SIMD 支持（std::simd），目前还是 nightly
- GPU 编程
  - 使用 CUDA 或 OpenCL 等框架，可以利用 GPU 的并行计算能力
  - 现代 GPU 的并行计算能力非常强大，可以显著提高计算性能

6.2.10 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。

6.2.11 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？

6.2.12 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。

6.2.13 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）。

6.2.14 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?

6.3 线程

6.3.1 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!");
});
```

6.3.2 线程句柄

thread::spawn 返回 JoinHandler 类型的线程句柄，用于后续对线程的管理和控制。

Rust

```rust
use std::thread;

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

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

6.3.3 线程句柄（续）

- 句柄的性质
  - 当线程的句柄被丢弃时，线程会变为分离（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）。
- join 函数
  - 会阻塞当前线程的执行，直到句柄对应的线程终止。
  - 返回对应线程返回值的 `Ok`，或者是发生恐慌时的参数的 `Err`。

6.3.4 线程与恐慌

- 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，和其他线程没有关系。
  - 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。

6.3.5 线程的操作

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。

```rust
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。

6.3.6 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!");
    });
}
```

6.3.7 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

```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
```

6.3.8 线程与所有权

- 这里，线程所执行的闭包生命周期超过了循环，因此要获得 i 的所有权。
- 用 move 表示通过转移捕获变量，获得所有权。

```rust
use std::thread;

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

6.4 共享线程状态

6.4.1 Send 与 Sync 特型

- Rust 的类型系统包含标注并发承诺的特型：
  - 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。

6.4.2 Send

```rust
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。

```rust
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) 是 Sync 的。
  - 所有不带内部可变性的类型会自动获得 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。

6.4.5 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。

6.4.6 共享线程状态示例

Rust

```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;
//         ^~~~
```

6.4.7 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。

- 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。

6.4.9 共享线程状态示例

Rust

```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));
}
```

6.4.10 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。

6.4.11 共享线程状态示例中的问题：互斥锁

- `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。

- 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。

6.4.13 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。

6.4.14 共享线程状态示例

Rust

```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));
}
```

6.5 通道

6.5.1 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。

6.5.2 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**。

- 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。

6.5.4 std::sync::mpsc

Rust

```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);
}
```

- 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。

- 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。

6.5.7 并发编程的难点：Rust 方案

- **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?

6.5.8 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。

6.6 小结

6.6.1 本讲小结

- 闭包
- 并发的概念
- 线程
- 共享线程状态
- 通道