第十四讲：信号量与管程

第 6 节：Rust 语言中的同步机制

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

1. 第 6 节：Rust 语言中的同步机制
   - 引用计数
   - 原子操作
   - 执行同步
   - 条件变量
   - 互斥信号量
   - 读写锁

Higher-level synchronization objects in Rust

- Arc: A thread-safe atomically Reference-Counted pointer, which can be used in multithreaded environments.
- Barrier: Ensures multiple threads will wait for each other to reach a point in the program, before continuing execution all together.
- Condvar: Condition Variable, providing the ability to block a thread while waiting for an event to occur.
- Mutex: Mutual Exclusion mechanism, which ensures that at most one thread at a time is able to access some data.
- RwLock: Provides a mutual exclusion mechanism which allows multiple readers at the same time, while allowing only one writer at a time.

Rc(Reference Counting)

A single-threaded reference-counting pointer. 'Rc' stands for 'Reference Counted'.

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<T>,
}
```

Example of Reference Counting

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

fn main() {
    let rc_examples = "Rc examples".to_string();
    {
        println!("--- rc_a is created ---");

        let rc_a: Rc<String> = Rc::new(rc_examples);
        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        {
            println!("--- rc_a is cloned to rc_b ---");

            let rc_b: Rc<String> = Rc::clone(&rc_a);
            println!("Reference Count of rc_b: {}", Rc::strong_count(&rc_b));
            println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

            // Two `Rc`s are equal if their inner values are equal
            println!("rc_a and rc_b are equal: {}", rc_a.eq(&rc_b));

            // We can use methods of a value directly
            println!("Length of the value inside rc_a: {}", rc_a.len());
            println!("Value of rc_b: {}", rc_b);

            println!("--- rc_b is dropped out of scope ---");
        }

        println!("Reference Count of rc_a: {}", Rc::strong_count(&rc_a));

        println!("--- rc_a is dropped out of scope ---");
    }

    // Error! `rc_examples` already moved into `rc_a`
    // And when `rc_a` is dropped, `rc_examples` is dropped together
    // println!("rc_examples: {}", rc_examples);
    // TODO ^ Try uncommenting this line
}
```

```
--- rc_a is created ---
Reference Count of rc_a: 1
--- rc_a is cloned to rc_b ---
Reference Count of rc_b: 2
Reference Count of rc_a: 2
rc_a and rc_b are equal: true
Length of the value inside rc_a: 11
Value of rc_b: Rc examples
--- rc_b is dropped out of scope ---
Reference Count of rc_a: 1
--- rc_a is dropped out of scope ---
```

Arc: Atomically Reference-Counted pointer

A thread-safe reference-counting pointer. 'Arc' stands for 'Atomically Reference Counted'.

```rust
pub struct Arc<T: ?Sized> {
    ptr: NonNull<ArcInner<T>>,
    phantom: PhantomData<ArcInner<T>>,
}
```

Methods in std::sync::Arc

```rust
pub fn new(data: T) -> Arc<T>
pub fn new_uninit() -> Arc<MaybeUninit<T>>
pub fn new_zeroed() -> Arc<MaybeUninit<T>>
pub fn pin(data: T) -> Pin<Arc<T>>
pub fn try_unwrap(this: Arc<T>) -> Result<T, Arc<T>>
pub fn new_uninit_slice(len: usize) -> Arc<[MaybeUninit<T>]>
pub unsafe fn assume_init(self) -> Arc<[T]>
pub fn into_raw(this: Arc<T>) -> *const T
pub unsafe fn from_raw(ptr: *const T) -> Arc<T>
pub fn into_raw_non_null(this: Arc<T>) -> NonNull<T>
pub fn downgrade(this: &Arc<T>) -> Weak<T>
pub fn weak_count(this: &Arc<T>) -> usize
pub fn strong_count(this: &Arc<T>) -> usize
pub fn ptr_eq(this: &Arc<T>, other: &Arc<T>) -> bool
pub fn make_mut(this: &mut Arc<T>) -> &mut T
pub fn get_mut(this: &mut Arc<T>) -> Option<&mut T>
pub unsafe fn get_mut_unchecked(this: &mut Arc<T>) -> &mut T
pub fn downcast<T>(self) -> Result<Arc<T>, Arc<dyn Any + 'static + Send + Sync>>
```

Atomic

Atomic types provide primitive shared-memory communication between threads, and are the building blocks of other concurrent types.

```
pub struct $atomic_type {
                v: UnsafeCell<$int_type>,
}
```

Methods in Atomic

```
impl AtomicUsize
pub const fn new(v: usize) -> Self
pub fn get_mut(&mut self) -> &mut usize
pub fn into_inner(self) -> usize
pub fn load(&self, order: Ordering) -> usize
pub fn store(&self, val: usize, order: Ordering)
pub fn swap(&self, val: usize, order: Ordering) -> usize
pub fn compare_and_swap
pub fn compare_exchange
pub fn compare_exchange_weak
pub fn fetch_add(&self, val: usize, order: Ordering) -> usize
pub fn fetch_sub(&self, val: usize, order: Ordering) -> usize
pub fn fetch_and(&self, val: usize, order: Ordering) -> usize
pub fn fetch_nand(&self, val: usize, order: Ordering) -> usize
pub fn fetch_or(&self, val: usize, order: Ordering) -> usize
pub fn fetch_xor(&self, val: usize, order: Ordering) -> usize
pub fn fetch_update<F>
pub fn fetch_max(&self, val: usize, order: Ordering) -> usize
pub fn fetch_min(&self, val: usize, order: Ordering) -> usize
pub fn as_mut_ptr(&self) -> *mut usize
```

Barrier

A barrier enables multiple threads to synchronize the beginning of some computation.

```
pub struct Barrier {
    lock: Mutex<BarrierState>,
    cvar: Condvar,
    num_threads: usize,
}

impl Barrier
pub fn new(n: usize) -> Barrier
pub fn wait(&self) -> BarrierWaitResult
```

Example of Barrier

```rust
#![allow(unused)]
fn main() {
use std::sync::{Arc, Barrier};
use std::thread;

let mut handles = Vec::with_capacity(10);
let barrier = Arc::new(Barrier::new(10));
for _ in 0..10 {
    let c = barrier.clone();
    // The same messages will be printed together.
    // You will NOT see any interleaving.
    handles.push(thread::spawn(move|| {
        println!("before wait");
        c.wait();
        println!("after wait");
    }));
}
// Wait for other threads to finish.
for handle in handles {
    handle.join().unwrap();
}
}
```

Execution Close

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.78s
     Running `target/debug/playground`
```

Standard Output

```
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
```

Condvar

Condition variables represent the ability to block a thread such that it consumes no CPU time while waiting for an event to occur.

```
pub struct Condvar {
    inner: Box<sys::Condvar>,
    mutex: AtomicUsize,
}

impl Condvar
pub fn new() -> Condvar
pub fn wait<'a, T>
pub fn wait_while<'a, T, F>
pub fn wait_timeout_ms<'a, T>
pub fn wait_timeout<'a, T>
pub fn wait_timeout_while<'a, T, F>
pub fn notify_one(&self)
pub fn notify_all(&self)
```

Example of Condvar

```rust
#![allow(unused)]
fn main() {
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = pair.clone();

thread::spawn(move|| {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    // We notify the condvar that the value has changed.
    println!("notify_all");
    cvar.notify_all();
});

// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
// As long as the value inside the `Mutex<bool>` is `false`, we wait.
while !*started {
    println!("before wait");
    started = cvar.wait(started).unwrap();
    println!("after wait");
}
}
```

Execution Close

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/playground`
```

Standard Output

```
before wait
notify_all
after wait
```

Mutex

A mutual exclusion primitive useful for protecting shared data.

```
pub struct Mutex<T: ?Sized> {
    // Note that this mutex is in a *box*, not inlined into the struct itself.
    // Once a native mutex has been used once, its address can never change (it
    // can't be moved). This mutex type can be safely moved at any time, so to
    // ensure that the native mutex is used correctly we box the inner mutex to
    // give it a constant address.
    inner: Box<sys::Mutex>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}

impl<T> Mutex<T>
pub fn new(t: T) -> Mutex<T>
pub fn lock(&self) -> LockResult<MutexGuard<T>>
pub fn try_lock(&self) -> TryLockResult<MutexGuard<T>>
pub fn is_poisoned(&self) -> bool
pub fn into_inner(self) -> LockResult<T>
pub fn get_mut(&mut self) -> LockResult<&mut T>
```

RwLock

Rwlock allows a number of readers or at most one writer at any point in time. The write portion of this lock typically allows modification of the underlying data (exclusive access).

```
pub struct RwLock<T: ?Sized> {
    inner: Box<sys::RWLock>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

Methods in Rwlock

```
impl<T: ?Sized> RwLock<T>
pub fn read(&self) -> LockResult<RwLockReadGuard<T>>
pub fn try_read(&self) -> TryLockResult<RwLockReadGuard<T>>
pub fn write(&self) -> LockResult<RwLockWriteGuard<T>>
pub fn try_write(&self) -> TryLockResult<RwLockWriteGuard<T>>
pub fn is_poisoned(&self) -> bool
pub fn into_inner(self) -> LockResult<T>
pub fn get_mut(&mut self) -> LockResult<&mut T>
```