part15.rs

Rust-101, Part 15: Mutex, Interior Mutability (cont.), RwLock, Sync

We already saw that we can use `Arc` to share memory between threads. However, `Arc` can only provide *read-only* access to memory: Since there is aliasing, Rust cannot, in general, permit mutation. To implement shared-memory concurrency, we need to have aliasing and permutation - following, of course, some strict rules to make sure there are no data races. In Rust, shared- memory concurrency is obtained through *interior mutability*, which we already discussed in a single-threaded context in part 12.

`Mutex`

The most basic type for interior mutability that supports concurrency is `Mutex<T>`. This type implements *critical sections* (or *locks*), but in a data-driven way: One has to specify the type of the data that's protected by the mutex, and Rust ensures that the data is *only* accessed through the mutex. In other words, "lock data, not code" is actually enforced by the type system, which becomes possible because of the discipline of ownership and borrowing.

As an example, let us write a concurrent counter. As usual in Rust, we first have to think about our data layout: That will be `Mutex<usize>`. Of course, we want multiple threads to have access to this `Mutex`, so we wrap it in an `Arc`.

Rather than giving every field a name, a struct can also be defined by just giving a sequence of types (similar to how a variant of an `enum` is defined). This is called a *tuple struct*. It is often used when constructing a *newtype*, as we do here: `ConcurrentCounter` is essentially just a new name for `Arc<Mutex<usize>>`. However, it is a locally declared types, so we can give it an inherent implementation and implement traits for it. Since the field is private, nobody outside this module can even know the type we are wrapping.

The derived `Clone` implementation will clone the `Arc`, so all clones will actually talk about the same counter.

The constructor just wraps the constructors of `Arc` and `Mutex`.

The core operation is, of course, `increment`.

`lock` on a mutex returns a guard, very much like `RefCell`. The guard gives access to the data contained in the mutex. (We will discuss the `unwrap` soon.) `.0` is how we access the first component of a tuple or a struct.

The guard is a smart pointer to the content.

At the end of the function, `counter` is dropped and the mutex is available again. This can only happen when full ownership of the guard is given up. In particular, it is impossible for us to take a reference to some of its content, release the lock of the mutex, and subsequently access the protected data without holding the lock. Enforcing the locking discipline is expressible in the Rust type system, so we don't have to worry about data races *even though* we are mutating shared memory!

One of the subtle aspects of locking is *poisoning*. If a thread panics while it holds a lock, it could leave the data-structure in a bad state. The lock is hence considered *poisoned*. Future attempts to `lock` it will fail. Above, we simply assert via `unwrap` that this will never happen. Alternatively, we could have a look at the poisoned state and attempt to recover from it.

The function `get` returns the current value of the counter.

Now our counter is ready for action.

We clone the counter for the first thread, which increments it by 2 every 15ms.

The second thread increments the counter by 3 every 20ms.

Now we watch the threads working on the counter.

Finally, we wait for all the threads to finish to be sure we can catch the counter's final value.

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

#[derive(Clone)]
struct ConcurrentCounter(Arc<Mutex<usize>>);

impl ConcurrentCounter {
    pub fn new(val: usize) -> Self {
        ConcurrentCounter(Arc::new(Mutex::new(val)))
    }
    pub fn increment(&self, by: usize) {
        let mut counter = self.0.lock().unwrap();
        *counter = *counter + by;
    }
    pub fn get(&self) -> usize {
        let counter = self.0.lock().unwrap();
        *counter
    }
}
pub fn main() {
    let counter = ConcurrentCounter::new(0);
    let counter1 = counter.clone();
    let handle1 = thread::spawn(move || {
        for _ in 0..10 {
            thread::sleep(Duration::from_millis(15));
            counter1.increment(2);
        }
    });
    let counter2 = counter.clone();
    let handle2 = thread::spawn(move || {
        for _ in 0..10 {
            thread::sleep(Duration::from_millis(20));
            counter2.increment(3);
        }
    });
    for _ in 0..50 {
        thread::sleep(Duration::from_millis(5));
        println!("Current value: {}", counter.get());
    }
    handle1.join().unwrap();
    handle2.join().unwrap();
    println!("Final value: {}", counter.get());
}
```

Exercise 15.1: Add an operation `compare_and_inc(&self, test: usize, by: usize)` that increments the counter by `by` *only if* the current value is `test`.

Exercise 15.2: Rather than panicking in case the lock is poisoned, we can use `into_inner` on the error to recover the data inside the lock. Change the code above to do that. Try using `unwrap_or_else` for this job.

`RwLock`

Besides `Mutex`, there's also `RwLock`, which provides two ways of locking: One that grants only read-only access, to any number of concurrent readers, and another one for exclusive write access. Notice that this is the same pattern we already saw with shared vs. mutable references. Hence another way of explaining `RwLock` is to say that it is like `RefCell`, but works even for concurrent access. Rather than panicking when the data is already borrowed, `RwLock` will of course block the current thread until the lock is available. In this view, `Mutex` is a stripped-down version of `RwLock` that does not distinguish readers and writers.

Exercise 15.3: Change the code above to use `RwLock`, such that multiple calls to `get` can be executed at the same time.

`Sync`

Clearly, if we had used `RefCell` rather than `Mutex`, the code above could not work: `RefCell` is not prepared for multiple threads trying to access the data at the same time. How does Rust make sure that we don't accidentally use `RefCell` across multiple threads?

In part 13, we talked about types that are marked `Send` and thus can be moved to another thread. However, we did *not* talk about the question whether a reference is `Send`. For `&mut T`, the answer is: It is `Send` whenever `T` is send. `&mut` allows moving values back and forth, it is even possible to `swap` the contents of two mutable references. So in terms of concurrency, sending a mutable, unique reference is very much like sending full ownership, in the sense that it can be used to move the object to another thread.

But what about `&T`, a shared reference? Without interior mutability, it would always be all- right to send such values. After all, no mutation can be performed, so there can be as many threads accessing the data as we like. In the presence of interior mutability though, the story gets more complicated. Rust introduces another marker trait for this purpose: `Sync`. A type `T` is `Sync` if and only if `&T` is `Send`. Just like `Send`, `Sync` has a default implementation and is thus automatically implemented for a data-structure *if* all its members implement it.

Since `Arc` provides multiple threads with a shared reference to its content, `Arc<T>` is only `Send` if `T` is `Sync`. So if we had used `RefCell` above, which is *not* `Sync`, Rust would have caught that mistake. Notice however that `RefCell` *is* `Send`! If ownership of the entire cell is moved to another thread, it is still not possible for several threads to try to access the data at the same time.

Almost all the types we saw so far are `Sync`, with the exception of `Rc`. Remember that a shared reference is good enough for cloning, and we don't want other threads to clone our local `Rc` (they would race for updating the reference count), so it must not be `Sync`. The rule of `Mutex` is to enforce synchronization, so it should not be entirely surprising that `Mutex<T>` is `Send` *and* `Sync` provided that `T` is `Send`.

You may be curious whether there is a type that's `Sync`, but not `Send`. There are indeed rather esoteric examples of such types, but that's not a topic I want to go into. In case you are curious, there's a Rust RFC, which contains a type `RcMut` that would be `Sync` and not `Send`. You may also be interested in this blog post on the topic.