所有权与智能指针

jask

09/27/2024

基于 Send 和 Sync 的线程安全

无法用于多线程的 Rc

例如：

```
use std::thread;
use std::rc::Rc;
fn main(){
  let v=Rc::new(5);
  let t=thread::spawn(move ||{
    println!("{}",v);
  });
  t.join().unwrap();
}
```

这段代码看似正确，实则报错。

因为 Rc 无法在线程间安全转移，也就是 Send 没有给 Rc 的实现。

Rc 和 Arc 的比较

```
// Rc 源码片段
impl<T: ?Sized> !marker::Send for Rc<T> {}
impl<T: ?Sized> !marker::Sync for Rc<T> {}

// Arc 源码片段
unsafe impl<T: ?Sized + Sync + Send> Send for Arc<T> {}
unsafe impl<T: ?Sized + Sync + Send> Sync for Arc<T> {}
```

! 代表移除特征的相应实现，上面代码中 Rc 的 Send 和 Sync 特征被特地移除了实现，而 Arc 则相反，实现了 Sync + Send，再结合之前的编译器报错，大概可以明白了：Send 和 Sync 是在线程间安全使用一个值的关键。

Send 和 Sync

Send 和 Sync 是 Rust 安全并发的重中之重，但是实际上它们只是标记特征 (marker trait，该特征未定义任何行为，因此非常适合用于标记)，来看看它们的作用：

实现Send的类型可以在线程间安全的传递其所有权

实现Sync的类型可以在线程间安全的共享(通过引用)

这里还有一个潜在的依赖：一个类型要在线程间安全的共享的前提是，指向它的引用必须能在线程间传递。因为如果引用都不能被传递，我们就无法在多个线程间使用引用去访问同一个数据了。

由上可知，若类型 T 的引用 &T 是 Send，则 T 是 Sync。

参考 RwLock 的实现：

```
unsafe impl<T: ?Sized + Send + Sync> Sync for RwLock<T> {}
```

首先 RwLock 可以在线程间安全的共享，那它肯定是实现了 Sync，但是我们的关注点不在这里。众所周知，RwLock 可以并发的读，说明其中的值 T 必定也可以在线程间共享，那 T 必定要实现 Sync。

果不其然，上述代码中，T 的特征约束中就有一个 Sync 特征，那问题又来了，Mutex 是不是相反？再来看看：

```rust
unsafe impl<T: ?Sized + Send> Sync for Mutex<T> {}
```

不出所料，Mutex 中的 T 并没有 Sync 特征约束。

实现 Send 和 Sync 的类型

在 Rust 中，几乎所有类型都默认实现了 Send 和 Sync，而且由于这两个特征都是可自动派生的特征（通过 derive 派生），意味着一个复合类型（例如结构体），只要它内部的所有成员都实现了 Send 或者 Sync，那么它就自动实现了 Send 或 Sync。

正是因为以上规则，Rust 中绝大多数类型都实现了 Send 和 Sync，除了以下几个（事实上不止这几个，只不过它们比较常见）：

裸指针两者都没实现，因为它本身就没有任何安全保证
UnsafeCell不是Sync，因此Cell和RefCell也不是
Rc两者都没实现（因为内部的引用计数器不是线程安全的）

当然，如果是自定义的复合类型，那没实现那哥俩的就较为常见了：只要复合类型中有一个成员不是 Send 或 Sync，那么该复合类型也就不是 Send 或 Sync。

手动实现 Send 和 Sync 是不安全的，通常并不需要手动实现 Send 和 Sync trait，实现者需要使用 unsafe 小心维护并发安全保证。