iPotato | Rust 中几个智能指针的异同与使用场景

ipotato.me/article/57

Rust 中几个智能指针的异同与使用场景

2020-01-16

想必写过 C 的程序员对指针都会有一种复杂的情感，与内存相处的过程中可以说是成也指针，败也指针。一不小心又越界访问了，一不小心又读到了内存里的脏数据，一不小心多线程读写数据又不一致了……我知道讲到这肯定会有人觉得“出这种问题还不是因为你菜”云云，但是有一句话说得好：“自由的代价就是需要时刻保持警惕”。

Rust 几乎把“内存安全”作为了语言设计哲学之首，从多个层面（编译，运行时检查等）极力避免了许多内存安全问题。所以比起让程序员自己处理指针（在 Rust 中可以称之为 Raw Pointer），Rust 提供了几种关于指针的封装类型，称之为智能指针（Smart Pointer），且对于每种智能指针，Rust 都对其做了很多行为上的限制，以保证内存安全。

- Box<T>
- Rc<T> 与 Arc<T>
- Cell<T>
- RefCell<T>

我在刚开始学习智能指针这个概念的时候有非常多的困惑，Rust 官方教程本身对此的叙述并不详尽，加之 Rust 在中文互联网上内容匮乏，我花了很久才搞清楚这几个智能指针封装的异同，在这里总结一下，以供参考，如有错误，烦请大家指正。

> 以下内容假定本文的读者了解 Rust 的基础语法，所有权以及借用的基本概念，这里是相关链接。

Box<T>

`Box<T>` 与大多数情况下我们所熟知的指针概念基本一致，它是一段指向堆中数据的指针。我们可以通过这样的操作访问和修改其指向的数据：

```
let a = Box::new(1);  // Immutable
println!("{}", a);    // Output: 1

let mut b = Box::new(1);  // Mutable
*b += 1;
println!("{}", b);    // Output: 2
```

然而 `Box<T>` 的主要特性是单一所有权，即同时只能有一个人拥有对其指向数据的所有权，并且同时只能存在一个可变引用或多个不可变引用，这一点与 Rust 中其他属于堆上的数据行为一致。

```
let a = Box::new(1);  // Owned by a
let b = a;  // Now owned by b

println!("{}", a);  // Error: value borrowed here after move

let c = &mut a;
let d = &a;

println!("{}, {}", c, d);  // Error: cannot borrow `a` as immutable because it is
also borrowed as mutable
```

Rc<T> 与 Arc<T>

`Rc<T>` 主要用于同一堆上所分配的数据区域需要有多个只读访问的情况，比起使用 `Box<T>` 然后创建多个不可变引用的方法更优雅也更直观一些，以及比起单一所有权，`Rc<T>` 支持多所有权。

`Rc` 为 Reference Counter 的缩写，即为引用计数，Rust 的 Runtime 会实时记录一个 `Rc<T>` 当前被引用的次数，并在引用计数归零时对数据进行释放（类似 Python 的 GC 机制）。因为需要维护一个记录 `Rc<T>` 类型被引用的次数，所以这个实现需要 Runtime Cost。

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

fn main() {
    let a = Rc::new(1);
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Rc::clone(&a);
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Rc::clone(&a);
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

输出依次会是 1 2 3 2。

需要注意的主要有两点。首先，`Rc<T>` 是完全不可变的，可以将其理解为对同一内存上的数据同时存在的多个只读指针。其次，`Rc<T>` 是只适用于单线程内的，尽管从概念上讲不同线程间的只读指针是完全安全的，但由于 `Rc<T>` 没有实现在多个线程间保证计数一致性，所以如果你尝试在多个线程内使用它，会得到这样的错误：

```
use std::thread;
use std::rc::Rc;

fn main() {
    let a = Rc::new(1);
    thread::spawn(|| {
        let b = Rc::clone(&a);
        // Error: `std::rc::Rc<i32>` cannot be shared between threads safely
    }).join();
}
```

如果想在不同线程中使用 `Rc<T>` 的特性该怎么办呢？答案是 `Arc<T>` ，即 Atomic reference counter。此时引用计数就可以在不同线程中安全的被使用了。

```
use std::thread;
use std::sync::Arc;

fn main() {
    let a = Arc::new(1);
    thread::spawn(move || {
        let b = Arc::clone(&a);
        println!("{}", b);  // Output: 1
    }).join();
}
```

Cell<T>

`Cell<T>` 其实和 `Box<T>` 很像，但后者同时不允许存在多个对其的可变引用，如果我们真的很想做这样的操作，在需要的时候随时改变其内部的数据，而不去考虑 Rust 中的不可变引用约束，就可以使用 `Cell<T>` 。 `Cell<T>` 允许多个共享引用对其内部值进行更改，实现了「内部可变性」。

```
fn main() {
    let x = Cell::new(1);
    let y = &x;
    let z = &x;
    x.set(2);
    y.set(3);
    z.set(4);
    println!("{}", x.get());  // Output: 4
}
```

这段看起来非常不 Rust 的 Rust 代码其实是可以通过编译并运行成功的， `Cell<T>` 的存在看起来似乎打破了 Rust 的设计哲学，但由于仅仅对实现了 `Copy` 的 `T` ， `Cell<T>` 才能进行 `.get()` 和 `.set()` 操作。而实现了 `Copy` 的类型在 Rust 中几乎等同于会分配在栈上的数据（可以直接按比特进行连续 n 个长度的复制），所以对其随意进行改写是十分安全的，不会存在堆数据泄露的风险（比如我们不能直接复制一段栈上的指针，因为指针指向的内容可能早已物是人非）。也是得益于 `Cell<T>` 实现了外部不可变时的内部可变形，可以允许以下行为的发生：

```
use std::cell::Cell;

fn main() {
    let a = Cell::new(1);

    {
        let b = &a;
        b.set(2);
    }


    println!("{:?}", a);  // Output: 2
}
```

如果换做 `Box<T>` ，则在中间出现的 Scope 就会使 `a` 的所有权被移交，且在执行完毕之后被 `Drop` 。最后还有一点， `Cell<T>` 只能在单线程的情况下使用。

RefCell<T>

因为 `Cell<T>` 对 `T` 的限制：只能作用于实现了 `Copy` 的类型，所以应用场景依旧有限（安全的代价）。但是我如果就是想让任何一个 `T` 都可以塞进去该咋整呢？ `RefCell<T>` 去掉了对 `T` 的限制，但是别忘了要牢记初心，不忘继续践行 Rust 的内存安全的使命，既然不能在读写数据时简单的 `Copy` 出来进去了，该咋保证内存安全呢？相对于标准情况的静态借用， `RefCell<T>` 实现了运行时借用，这个借用是临时的，而且 Rust 的 Runtime 也会随时紧盯 `RefCell<T>` 的借用行为：同时只能有一个可变借用存在，否则直接 Painc。也就是说 `RefCell<T>` 不会像常规时一样在编译阶段检查引用借用的安全性，而是在程序运行时动态的检查，从而提供在不安全的行为下出现一定的安全场景的可行性。

```
use std::cell::RefCell;
use std::thread;

fn main() {
    thread::spawn(move || {
       let c = RefCell::new(5);
       let m = c.borrow();

       let b = c.borrow_mut();
    }).join();
    // Error: thread '<unnamed>' panicked at 'already borrowed: BorrowMutError'
}
```

如上程序所示，如同一个读写锁应该存在的情景一样，直接进行读后写是不安全的，所以 `borrow` 过后 `borrow_mut` 会导致程序 Panic。同样， `ReCell<T>` 也只能在单线程中使用。

如果你要实现的代码很难满足 Rust 的编译检查，不妨考虑使用 `Cell<T>` 或 `RefCell<T>` ，它们在最大程度上以安全的方式给了你些许自由，但别忘了时刻警醒自己自由的代价是什么，也许获得喘息的下一秒，一个可怕的 Panic 就来到了你身边！

组合使用

如果遇到要实现一个同时存在多个不同所有者，但每个所有者又可以随时修改其内容，且这个内容类型 `T` 没有实现 `Copy` 的情况该怎么办？使用 `Rc<T>` 可以满足第一个要求，但是由于其是不可变的，要修改内容并不可能；使用 `Cell<T>` 直接死在了 `T` 没有实现 `Copy` 上；使用 `RefCell<T>` 由于无法满足多个不同所有者的存在，也无法实施。可以看到各个智能指针可以解决其中一个问题，既然如此，为何我们不把 `Rc<T>` 与 `RefCell<T>` 组合起来使用呢？

```
use std::rc::Rc;
use std::cell::RefCell;

fn main() {
    let shared_vec: Rc<RefCell<_>> = Rc::new(RefCell::new(Vec::new()));
    // Output: []
    println!("{:?}", shared_vec.borrow());
    {
        let b = Rc::clone(&shared_vec);
        b.borrow_mut().push(1);
        b.borrow_mut().push(2);
    }
    shared_vec.borrow_mut().push(3);
    // Output: [1, 2, 3]
    println!("{:?}", shared_vec.borrow());
}
```

通过 `Rc<T>` 保证了多所有权，而通过 `RefCell<T>` 则保证了内部数据的可变性。

参考

Tagged in : Rust Programming Language Smart Pointer Explain